Logical Analysis of Biological Data

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von

Katinka Becker

Berlin, Januar 2020

1. Gutachter:
Prof. Dr. Alexander Bockmayr
Freie Universität Berlin
2. Gutachter:
Prof. Dr. Torsten Schaub
Universität Potsdam

Datum der Disputation: 15.07.21

Abstract

Methods and models of machine learning have become indispensable for the analysis and interpretation of big data sets in the biomedical field. This work focuses on the machine learning method *Logical Analysis of Data* (LAD), which combines concepts from combinatorics, Boolean functions and optimization. LAD is based on the generation of *patterns*. These patterns are used to communicate relevant information and form *theories*, which are the classifiers for the prediction of new data points. This thesis makes contributions to practice, theory and applications of LAD.

With regards to LAD practice, we present the design and development of our freely available software package `AnswerSetLAD`. In the implementation we make use of the declarative programming paradigm *Answer Set Programming* (ASP), which is oriented towards difficult combinatorial search and optimization problems. For that reason, it provides a perfectly suited framework for the LAD functionalities. In this thesis, we substantiate this statement with an empirical study on the running time of our ASP approach and a state-of-the-art Mixed-Integer Linear Programming (MILP) approach for the generation of maximal patterns, which are a specific type of LAD patterns.

We present two theoretical advancements of LAD concerning prime patterns. This pattern type plays a key role in the LAD method. Firstly, we propose an algorithm for the enumeration of all prime patterns of a data set. The algorithm is preferable to classical methods in the case that the data set has a small maximal Hamming distance between the two classes of data points. Secondly, we investigate theories formed of prime patterns. Since the number of such *prime theories* for a data set is large in general, we define a statistical measure that can be used to rank prime patterns and, based on this, select those prime patterns that are more significant than others to form a theory.

Finally, we illustrate two biological applications of LAD. In the first application, we use prime patterns to successfully identify protein interactions in a cell signaling network based on perturbation measurement data. The second application is located in the field of synthetic biology. Here we outline our approach of Boolean classifier generation out of miRNA data. These classifiers can be used for the assembly of in-vitro synthetic cell circuits to distinguish healthy from cancerous tissue.

Zusammenfassung

Maschinelle Lernverfahren und Modelle sind für die Analyse und Interpretation großer Datensätze im biomedizinischen Bereich unverzichtbar geworden. In dieser Arbeit befassen wir uns mit der *Logischen Analyse von Daten* (LAD), einer Methode für maschinelles Lernen. LAD vereint Konzepte aus Kombinatorik, Booleschen Funktionen und Optimierung und basiert auf der Erzeugung von *Mustern*. Diese Muster werden dazu verwendet relevante Informationen zu kommunizieren und *Theorien* zu bilden, welche dazu genutzt werden können Vorhersagen für neue Datenpunkte zu treffen. Diese Arbeit liefert Beiträge zu Praxis, Theorie und Anwendungen von LAD.

Im praktischen Teil der Arbeit stellen wir das Design und die Entwicklung unseres frei zugänglichen Software-Pakets `AnswerSetLAD` vor. Für die Implementierung nutzen wir das deklarative Programmierparadigma *Answer Set Programming* (ASP), welches auf schwierige kombinatorische Such- und Optimierungsprobleme ausgerichtet ist. Aus diesem Grund bietet es einen perfekt geeigneten Rahmen für die Implementierung der LAD Funktionalitäten. Wir untermauern diese Aussage innerhalb dieser Arbeit mit einem empirischen Vergleich zur Laufzeit unseres ASP und eines aktuell gebräuchlichen Mixed-Integer Linear Programming (MILP) Ansatzes zur Erzeugung maximaler Muster, einem speziellen Typ von LAD Mustern.

Wir präsentieren zwei theoretische Weiterentwicklungen bezüglich Primmustern in LAD. Dieser Mustertyp spielt eine zentrale Rolle in der LAD Methode. Zunächst schlagen wir einen Algorithmus zur Aufzählung aller Primmuster eines Datensatzes vor, welcher besonders in dem Fall eine geringere Laufzeit als klassische Methoden aufweist, in dem der Datensatz einen kleinen maximalen Hamming-Abstand zwischen den beiden Klassen von Datenpunkten hat. Außerdem untersuchen wir Theorien, die aus Primmustern aufgebaut sind. Die Anzahl solcher *Primtheorien* für einen Datensatz ist im Allgemeinen groß. Wir definieren ein statistisches Maß, das es ermöglicht, bestimmte sinnvolle Theorien aus der Menge der Primtheorien auszuwählen.

Abschließend zeigen wir zwei biologische Anwendungen von LAD. In der ersten Anwendung nutzen wir erfolgreich Primmuster um die Proteininteraktionen in einem Zellsignalnetzwerk basierend auf einem Perturbationsexperiment zu bestimmen. Die zweite Anwendung ist in der synthetischen Biologie angesiedelt. Hier erläutern wir unseren Ansatz zur Bestimmung Boolescher Klassifikatoren aus miRNA Daten. Diese Klassifikatoren können anschließend dazu genutzt werden um in-vitro synthetische Zellschaltkreise zu erzeugen, die gesundes von krebsbefallenem Gewebe unterscheiden können.

Contents

Part I

Background

1 Introduction

Many problems in biomedical research can be represented by the same question, namely “Which of the given attributes or combinations of attributes are responsible for a certain property of the observations in the data set?”. Here the attributes could be anything from measurements of protein activations or metabolic concentrations to miRNA profiles. The observation properties could be the activation or non-activation of a downstream protein or a disease, which is present or not. Such tasks belong to the area of classification problems.

Machine learning methods for data classification are a large and well investigated field of research. Support vector machines [32, 95], neural networks [19, 92, 39] and decision trees [85] are some of the most prominent approaches. While theoretical machine learning methods are a research topic on their own, their application in life sciences becomes increasingly important. Nowadays the technological progress and with it the availability of large data sets poses a major challenge to the research field of biology and medicine. This increase of data volume gives us, on the one hand, the chance to comprehend more about the underlying systems, while, on the other hand, it makes it more difficult to filter out reliably the relevant information. Data analysis, therefore, became a central part of life science over the last decades and systems biology formed a new growing field of research [63, 10]. Next to developing the methods for automated data analysis in the scope of systems biology, one of the major goals is that the insights that are derived from the data are communicated in a way that makes them understandable and valuable for the whole interdisciplinary field.

A machine learning method that combines both the capability of handling large data sets and the benefit of providing understandable information, is the *Logical Analysis of Data* (LAD) [35]. LAD builds on the generation of patterns and uses them to extract short pieces of information from the data.

In this thesis, we consider the Logical Analysis of Data and its application to biological data sets. We present our software package `AnswerSetLAD`, which provides the user with all functionalities of the LAD method and is available on GitHub [14]. It is designed making use of the declarative programming paradigm *Answer Set Programming* (ASP), which suits the problem of pattern enumeration perfectly. Besides the development of our software we present new theoretical ideas based on the LAD method. Furthermore, we show how patterns and classifiers according to LAD can be used to answer questions in certain biological fields like signaling networks and synthetic biology. We thereby make a contribution to analysis techniques in systems biology.

1.1 Aims and structure of this thesis

This thesis makes contributions to the methodology of *Logical Analysis of Data* (LAD) with respect to the practice, theory and application. The following paragraphs illustrate the contents of my thesis. It is organized in five parts being *Background*, *Practice*, *Theory*, *Application* and *Conclusion*, which represent the fields of contributions.

In this first chapter of the thesis, we give an introduction to the theoretical background of data analysis and its historical development. We talk briefly about commonly used methods for machine learning and classification. To prepare for the following work, we introduce Boolean functions and partially defined Boolean functions (pdBf), which form the basis for the method we apply.

The second chapter is an introduction to the methodology of *Logical Analysis of Data* (LAD) [35]. LAD is a machine learning technique, which combines ideas from combinatorics, Boolean functions and optimization. We introduce the basic workflow, namely *data binarization*, *pattern generation* and *theory formation*. In addition, we outline the development of the methodology from the original proposal in the 1980s until today.

In the third chapter, we present the design, implementation and performance of our software package `AnswerSetLAD`, which enables the user to apply the LAD functionalities to a (biological) data set. As programming framework we use *Answer Set Programming* (ASP) [73], a declarative programming paradigm oriented towards difficult search and combinatorial optimization problems. It suits the given tasks of pattern generation perfectly and is hence able to outperform the state-of-the-art Mixed-Integer Linear Programming (MILP) approaches as we show on the example of a specific pattern type. Besides the explanation of our software an introduction to the use of ASP is given.

The fourth chapter is based on theoretical ideas on the extension of the LAD methodology and focuses on prime patterns. We introduce a new algorithm, called `PrimePatternForest`, for the efficient calculation of prime patterns for data sets with the property of having small Hamming distance between the set of positive and negative observations. Secondly, we describe an approach to select a subset of theories from the large set of prime theories.

Chapter five focuses on biological applications and it is organized in two main sections, which both include work that has been published previously [15, 17]. In the first section, we show how `AnswerSetLAD` can be used on perturbation data sets of regulatory signaling networks to help understanding the underlying network structure. The second section displays our work in synthetic biology. Here Boolean functions are used to classify tissue data as healthy or cancerous depending on its miRNA profile. These so-called *cell classifier circuits* are of particular interest regarding personalized medicine.

The last chapter, Chapter six, gives a conclusion and future directions on the work described in this thesis.

1.2 Data classification and analysis

Current technological development helps to capture an increasing amount of information, but leads to a new challenge in data analysis. While the ongoing progress in experimental technologies helps us to treasure up more and more information, which might lead to a deeper understanding of the systems we are looking at, the work of analyzing the resulting data becomes more challenging. In this thesis, we focus on the analysis of data sets that can be partitioned into two classes, which we call *positive* and *negative observations*. Given a set of *observations* Ω consisting of *positive observations* Ω^+ and *negative observations* Ω^-, with $\Omega^+ \cap \Omega^- = \emptyset$, the goal of a binary *classification* problem is to decide for a new observation $X \notin \Omega$ whether it belongs to the positive or the negative class. This decision is called *prediction*.

At this point, it is worth noting that in the described scenario, where we have knowledge about a system based on some observations we made and want to make decisions for “unseen observations”, we will never actually know whether the *classifier* we build or the prediction we make is really true or false before we have seen the outcome of all missing observations. We can certainly define measures on *how well a classifier works*, which are usually based on the accuracy of a *test classifier*, which is generated by splitting the observed data into a *training set* and a *test set*. The classifier is then calculated using only the training set and its *accuracy* is measured on the test set in terms of how many predictions of the classifier were correct. For the application though, the classifier is built out of all data available and

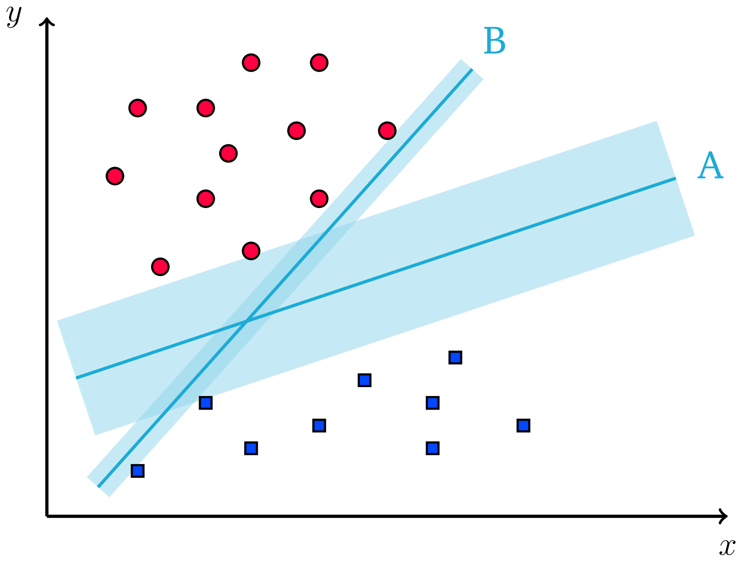


Fig. 1.1.: The *support vector machine* (SVM) is a large margin classifier, which aims to calculate a hyperplane that separates the two classes with a large margin. Hyperplane A would be chosen over hyperplane B in this visual example because its margin to the two classes is larger.

the outcome of a prediction is unclear until we have seen the actual result of the system.

While there exist some widely used measures for classifiers like the already mentioned accuracy, it is reasonable to ask whether these measures are an indication for a *good* classifier or if there are other properties that might be of even higher use.

In this thesis, we describe a machine learning and classification approach based on Boolean functions and combinatorics that is called *Logical Analysis of Data* (LAD) and was introduced by Peter L. Hammer [35]. The field of classification problems has been extensively studied and a lot of well known methods exist. LAD shows equal or even superior performance on classical data sets and carries benefit regarding applications in particular.

One of the main advantages of the analysis with LAD over other well-known machine learning methods, such as *Support Vector Machines* [103, 32, 95], *Decision Trees* [23, 85] or *Nearest Neighbor Search* [11], is the generation of *patterns*. Patterns are the main concept of LAD and extract the important information from the given data. They are easier to interpret and, therefore, easier to communicate than, e.g., hyperplanes, which result from a classification via a support vector machine (see Figure 1.1 for an intuition).

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

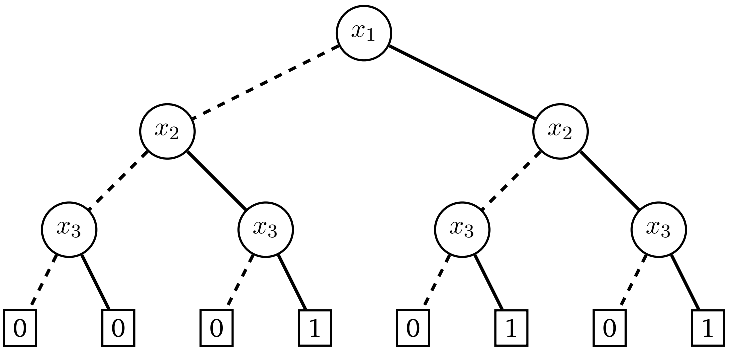


Fig. 1.2.: An example of a Boolean function and its representation as a Decision Tree.

In [23] the authors placed the LAD methodology in context with Decision Trees. Decision Trees [85] are binary trees. Starting at the root one walks along the graph choosing the left or right branch at the current node depending on whether the associated attribute is 1 or 0. When reaching a leaf the classification of the vector is given by the labeling of the leaf. An example of a Boolean function and a representation of it as a Decision Tree is shown in Figure 1.2. Every Boolean function can be represented by a Decision Tree. The authors in [23] show that a special class that they call *reasonable Decision Trees* is a sub class of so-called *bi-theories*, which are part of the LAD methodology and about which we will learn more in the next chapters.

LAD provides the possibility to *justify* the result on the given data [23]. This is different for most standard classification techniques. Not only can performance be measured in cross-validation tests or a posteriori clinical trial but the patterns themselves yield the justification of the generated result. These patterns form a *theory* to make predictions and are indicators of how well the theory suits the problem. Chapter 2 gives an overview of the LAD methodology and describes the mentioned concepts in more detail. Before starting with the introduction of the LAD notations and definitions, we present some preliminaries that are useful for the further understanding.

1.3 Preliminaries

We focus our work on binary data. Therefore, the basis for the following discussion will be the binary set $\mathbb{B} = \{0, 1\}$.

(x_1, x_2, x_3)	$g(x_1, x_2, x_3)$
$(0, 0, 0)$	0
$(1, 0, 0)$	1
$(0, 1, 0)$	0
$(0, 0, 1)$	1
$(1, 1, 0)$	1
$(1, 0, 1)$	1
$(0, 1, 1)$	0
$(1, 1, 1)$	1

Fig. 1.3.: A truth table of a Boolean function.

1.3.1 Boolean functions

The following definitions and notations are taken from [34], which is a standard reference in the field.

Definition 1.1 (Boolean function). *A* Boolean function *f of $n \in \mathbb{N} \setminus \{0\}$ variables is a function on $\mathbb{B}^n$ into $\mathbb{B}$, where $\mathbb{B}$ is the set $\{0, 1\}$ and $\mathbb{B}^n$ denotes the n-fold Cartesian product of $\mathbb{B}$ with itself. A point $X = (x_1, \dots, x_n) \in \mathbb{B}^n$ is a* true point *(*false point*) of f if $f(X) = 1$ ($f(X) = 0$). We denote with $T(f)$ the set of true points of f and with $F(f)$ the set of false points of f.*

There are different ways of defining a Boolean function. The most elementary way is the definition by a truth table.

Definition 1.2 (Truth table). *The* truth table *of a Boolean function on $\mathbb{B}^n$ is a complete list of all points in $\mathbb{B}^n$ together with the value of the function at each point.*

An example is given in Figure 1.3.

Boolean functions find various applications, e.g. in logic, combinatorics or game theory. An interpretation from electrical engineering are switching circuits. Those circuits can be modeled as directed acyclic graphs $D = (V, E)$ where the vertices V are different gates, which can be AND, OR or NOT gates. There are *input gates* $v_1, \dots, v_n$ having in-degree 0 and a single *output gate* g having out-degree 0. In Figure 1.4 the switching circuit belonging to the Boolean function of the truth table in Figure 1.3 is shown.

The same Boolean function is visualized via a geometric interpretation in Figure 1.5. Here the true points and false points of the Boolean function are represented by points in the n-dimensional hypercube.

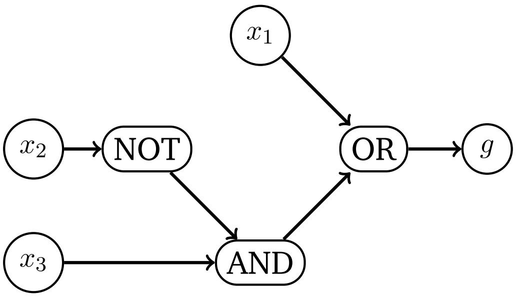


Fig. 1.4.: Switching circuit of the Boolean function defined by the truth table in Figure 1.3.

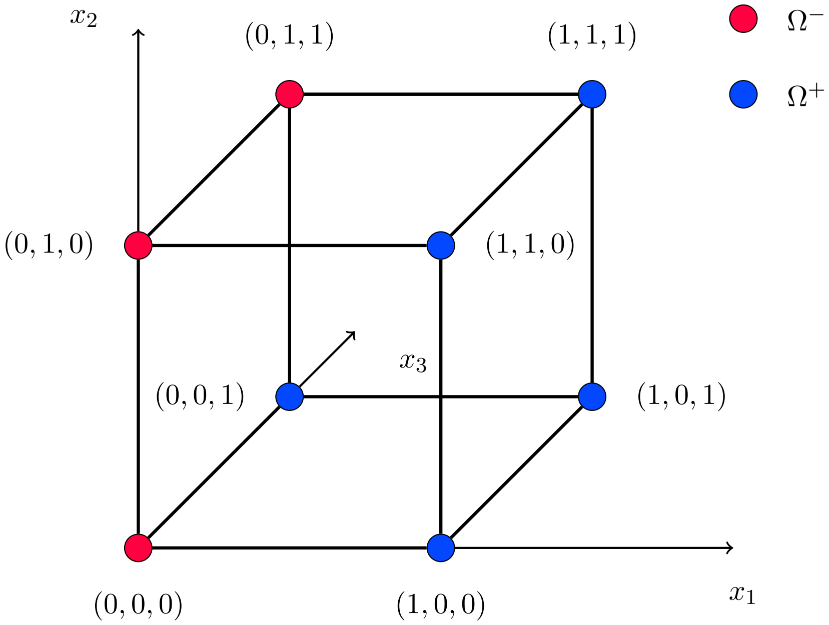


Fig. 1.5.: A geometric representation of the Boolean function in Figure 1.3.

Besides truth tables, directed graphs and cubes in $\mathbb{B}^n$ every Boolean function can be formulated using *Boolean expressions*. Before introducing the concept of Boolean expressions we define the three binary operations in the following.

Definition 1.3 (Binary operations)**.** *The Boolean OR* $\vee$ *(disjunction), the Boolean AND* $\wedge$ *(conjunction) and the Boolean NOT* $^-$ *are defined on* $\mathbb{B}$ *by the following rules:*

$$\begin{aligned} &0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1, \\ &0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1, \\ &\overline{1} = 0, \quad \overline{0} = 1. \end{aligned}$$

Definition 1.4 (Boolean expression)**.** *Let* $x_1, \ldots, x_n$ *be a finite set of Boolean variables. A* Boolean expression *on* $x_1, \ldots, x_n$ *is defined recursively by the following:*

1. *The constants 0,1 and the variables* $x_1, \ldots, x_n$ *are Boolean expressions in the variables* $x_1, \ldots, x_n$.
2. *If* Φ *and* Ψ *are Boolean expressions in* $x_1, \ldots, x_n$ *then* $\Phi \vee \Psi$*,* $\Phi \wedge \Psi$ *and* $\overline{\Phi}$ *are Boolean expressions in* $x_1, \ldots, x_n$.
3. *Every Boolean expression is formed by finitely many applications of the first two rules.*

We say that a Boolean expression in $x_1, \ldots, x_n$ *is a Boolean expression on* $\mathbb{B}^n$.

From this definition, we define in the next step how a Boolean function is represented by a Boolean expression.

Definition 1.5 (Boolean function)**.** *The Boolean function* f_Φ *represented by the Boolean expression* Φ *is the unique Boolean function on* $\mathbb{B}^n$ *defined as follows. For every point* $(x_1^*, \ldots, x_n^*) \in \mathbb{B}^n$ *the value of* $f(x_1^*, \ldots, x_n^*)$ *is obtained by substituting* x_i^* *for* x_i $(i = 1, \ldots, n)$ *in* Φ *and applying recursively the rules of the binary operations to compute the value of the resulting expression. When* $f = f_\Phi$ *on* $\mathbb{B}^n$ *we say that* f *admits the expression* Φ *and write* $f = \Phi$.

The Boolean function shown in the truth table given in Figure 1.3 can be represented by the following Boolean expression:

$$g(x_1, x_2, x_3) = (\overline{x_2} \wedge x_3) \vee (x_1).$$

It is important to notice that every Boolean function can be represented by various Boolean expressions, but a Boolean expression represents a *unique* Boolean function.

In fact, there exist 2^{2^n} Boolean functions of n variables while there are infinitely many Boolean expressions in n variables.

Normal forms

In order to make it more easy to compare Boolean functions represented by Boolean expressions, we introduce the concept of normal forms.

For a family of Boolean expressions $\{\Phi_k \mid k \in \Delta\}$ indexed over the set $\Delta = \{k_1, k_2, \ldots, k_m\}$ we denote by $\bigvee_{k\in\Delta} \Phi_k$ the expression $(\Phi_{k_1} \vee \Phi_{k_2} \vee \cdots \vee \Phi_{k_m})$ and we denote by $\bigwedge_{k\in\Delta} \Phi_k$ the expression $(\Phi_{k_1} \wedge \Phi_{k_2} \wedge \cdots \wedge \Phi_{k_m})$. By convention, when Δ is empty, $\bigvee_{k\in\Delta} \Phi_k$ is equivalent to the constant 0 and $\bigwedge_{k\in\Delta} \Phi_k$ is equivalent to the constant 1.

Definition 1.6 (Disjunctive and conjunctive normal form)**.** *Let x be a Boolean variable. A* literal *is an expression of the form x or $\overline{x} = 1 - x$. An* elementary conjunction *or* term *is an expression of the form*

$$C = \bigwedge_{i\in A} x_i \wedge \bigwedge_{j\in B} \overline{x_j}, \quad where A \cap B = \emptyset.$$

An elementary disjunction *or* clause *is an expression of the form*

$$D = \bigvee_{i\in A} x_i \vee \bigvee_{j\in B} \overline{x_j}, \quad where A \cap B = \emptyset.$$

A disjunctive normal form (DNF) *is an expression of the form*

$$\bigvee_{k=1}^{m} C_k = \bigvee_{k=1}^{m} (\bigwedge_{i\in A_k} x_i \wedge \bigwedge_{j\in B_k} \overline{x_j}),$$

where each C_k, $k = 1, \ldots, m$ is an elementary conjunction. We say that each conjunction C_k is a term *of the DNF.*

A conjunctive normal form (CNF) *is an expression of the form*

$$\bigwedge_{k=1}^{m} D_k = \bigwedge_{k=1}^{m} (\bigvee_{i\in A_k} x_i \vee \bigvee_{j\in B_k} \overline{x_j}),$$

where each D_k, $k = 1, \ldots, m$ is an elementary disjunction. We say that each disjunction D_k is a clause *of the CNF.*

A fundamental property of Boolean functions is captured in the following theorem [34].

Theorem 1.1. *Every Boolean function can be represented by a disjunctive normal form and a conjunctive normal form.*

Prime implicants

A field of Boolean functions, which has been intensively studied, is the field of *prime implicants*. Willard V. O. Quine [86] introduced prime implicants as minimal implicants of a Boolean function. Since we will introduce *prime patterns* as an extension of prime implicants in the context of partially defined Boolean functions in the later chapters, we here give a short introduction to prime implicants.

Definition 1.7. *Given two Boolean functions f and g on $\mathbb{B}^n$, we say that f implies g if*

$$f(X) = 1 \Rightarrow g(X) = 1 \; \forall X \in \mathbb{B}^n.$$

Definition 1.8 (Implicant)**.** *Let f be a Boolean function and C an elementary conjunction. We say that C is an* implicant *of f if C implies f.*

Definition 1.9 (Prime implicant)**.** *Let f be a Boolean function and C be an implicant of f. We say that C is a* prime implicant *of f if each elementary conjunction obtained by eliminating an arbitrary literal from it is not an implicant.*

Theorem 1.2. *[34] Every Boolean function can be represented by the disjunction of all its prime implicants.*

1.3.2 Partially defined Boolean functions

In this thesis, we put a special emphasis on *partially defined Boolean functions* (pdBf). These are Boolean functions that are defined on a subset of the points in $\{0, 1\}^n$ only.

Let $\Omega \subseteq \{0, 1\}^n$ be a set of binary points that we call *observations*. Let Ω be partitioned into two disjoint classes such that the set $\Omega = \Omega^+ \uplus \Omega^-$ is the disjoint union of Ω^+ and Ω^-, called *positive* and *negative observations*, respectively.

Definition 1.10 (Partially defined Boolean function (pdBf))**.** *Given a set $\Omega = \Omega^+ \uplus \Omega^-$ of positive and negative observations we call the pair (Ω^+, Ω^-) a* partially defined Boolean function (pdBf) *on $\mathbb{B}^n = \{0, 1\}^n$.*

A pdBf (Ω^+, Ω^-) in n variables is defined on a subset $\Omega \subseteq \{0,1\}^n$. By assigning 0 or 1 to each of the n-dimensional vectors of $\{0,1\}^n \setminus \Omega$, we can define an *extension* for (Ω^+, Ω^-).

Definition 1.11 (Extension)**.** *For a pdBf* (Ω^+, Ω^-) *on* $\mathbb{B}^n$ *a Boolean function* $e : \mathbb{B}^n \to \mathbb{B}$ *satisfying*

$$\Omega^+ \subseteq \Omega^+(e) \text{ and } \Omega^- \subseteq \Omega^-(e)$$

is called an extension *of* (Ω^+, Ω^-)*, where*

$$\Omega^+(e) = \{P \in \mathbb{B}^n \mid e(P) = 1\},$$
$$\Omega^-(e) = \{N \in \mathbb{B}^n \mid e(N) = 0\}.$$

Logical Analysis of Data | 2

In this chapter, we introduce the concepts and definitions of *Logical Analysis of Data* (LAD) [2, 35]. LAD is a methodology for data analysis that combines ideas from combinatorics and Boolean functions as well as from machine learning and optimization. It was introduced by Peter L. Hammer in a lecture given in 1986 and extended and published afterwards together with Yves Crama and Toshihide Ibaraki in [35].

2.1 Introduction

This thesis is located at the intersection of mathematics and computer science with biology and medicine. As described in the last chapter we aim to analyze biomedical data sets making use of a machine learning approach. *Logical Analysis of Data* (LAD) [2, 35] is a method combining ideas from various fields like Boolean functions, optimization and machine learning. It provides elaborated tools for data analysis.

LAD is based on the classical concepts of switching circuits and partially defined Boolean functions (pdBf). Since Boolean functions are a well-investigated research field and were already at the time when LAD was founded, a broad basis for LAD existed from the beginning. Over the last decades LAD has been further developed. For recent insights see [69, 31]. The overall research interest shifted from the methodology and implementation to the applications in various research fields, where the biomedical field attracted particular interest [1, 6, 67].

In the following sections we want to introduce the reader to the definitions and existing concepts of the LAD methodology.

2.1.1 An introductory example

We start the exploration of the LAD methodology with an introductory example adapted from [35].

Day	Apple	Bread	Chocolate	Cheese	Pasta	Broccoli	Stomach ache
1	0	0	1	0	1	1	Yes
2	0	0	0	1	1	0	No
3	1	1	1	1	0	1	Yes
4	0	1	1	0	0	1	Yes
5	1	1	0	0	0	1	No
6	0	1	0	1	1	0	Yes
7	1	0	1	0	1	1	No

Tab. 2.1.: List of food items the patient consumed over a day for one week.

A patient is complaining to his doctor about a stomach ache, which appears on some days and does not appear on others. After a physical examination the doctor comes to the assumption that the stomach ache could be caused by the food that his patient consumes over the day. Therefore, the doctor requests the patient to make a list of the different food items he eats over a day for one week.

One week later the patient comes back to see the doctor. The list he wrote down can be seen in Table 2.1. From the list of food items consumed over the days the physician concludes that on some days on which the patient had a stomach ache, he ate *chocolate* without eating an *apple*. He never did this on the days without stomach ache. Furthermore the doctor reveals that on some of the days with stomach ache the patient had bread and cheese, which he never had in this combination on the days without stomach ache. The clever doctor additionally realizes that those two combinations of food items or *patterns* explain all days with stomach ache in the list in Table 2.1. His *theory*, therefore, is that, to prevent the aching stomach, the patient should not eat these food item combinations.

In the following we want to generalize the approach of the doctor and for this reason we take a look at the questions that he had to ask himself in order to come up with his theory:

- How can we compute a short list of food items that explains the stomach ache? In our example *apple*, *chocolate*, *bread* and *cheese* were sufficient for this purpose.

- How can we detect patterns (i.e. combinations of food items) in the data set?

- What is a reasonable way to build theories (i.e. collections of patterns) explaining the observed stomach ache?

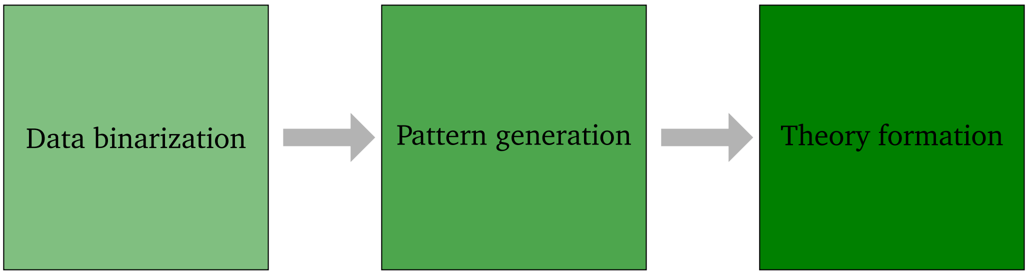


Fig. 2.1.: The three main steps of the LAD method.

In the following sections we introduce the formal concept and notations of LAD that are in line with these three questions.

2.1.2 The main steps of the LAD procedure

The LAD procedure can be divided into three main steps. These are *data binarization*, *pattern generation* and *theory formation*.

Within the following pages we introduce each of the three parts of the LAD method. Based on the original foundations of LAD, which was designed for binary data, we will, however, consider the binarization step of non-binary data last and proceed with the pattern generation and theory formation step first assuming that the underlying data is in binary form.

2.2 Basic concepts and notations

LAD was originally designed for binary data sets. In this section, we introduce notations on the basis of binary data. Since data in general is not binary the discretization step plays a key role in LAD and is, therefore, firmly established in the LAD workflow. We will talk about the binarization of numerical data according to LAD in Section 2.5.

Let $\Omega \subseteq \{0,1\}^n$ be a set of *observations* that is divided into two subsets by a *decision variable* x_0, such that $\Omega = \Omega^+ \uplus \Omega^-$ is the disjoint union of Ω^+ and Ω^-, called *positive* and *negative observations*, respectively (see Table 2.2). Note that the decision variable could be any of the given variables, each leading to a corresponding division in positive and negative observations.

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	
1	1	0	0	1	0	1	1	Ω^+
2	1	1	1	1	1	0	1	
3	1	0	1	1	0	0	1	
4	1	0	1	0	1	1	0	
5	0	0	0	0	1	1	0	Ω^-
6	0	1	1	0	0	0	1	
7	0	1	0	1	0	1	1	

Tab. 2.2.: Binary data set Ω partitioned into positive observations Ω^+ and negative observations Ω^- by decision variable x_0.

We recall the definition of a *Boolean function* of $n \in \mathbb{N}$ variables (Chapter 1 Definition 1.1). A Boolean function of $n \in \mathbb{N}$ variables is a mapping $f\colon \{0,1\}^n \to \{0,1\}$. There exist 2^{2^n} Boolean functions of n variables. A data set Ω as described above is a *partially defined Boolean function* (pdBf), meaning that the function f is only given for $\Omega = \Omega^+ \uplus \Omega^-$. Every function $e\colon \{0,1\}^n \to \{0,1\}$ with $e(x) = f(x)$ for all $x \in \Omega$ is called *extension* of f.

The two extreme extensions of a pdBf $f = (\Omega^+, \Omega^-)$, which we denote by Φ_f^+ and Φ_f^- are defined by

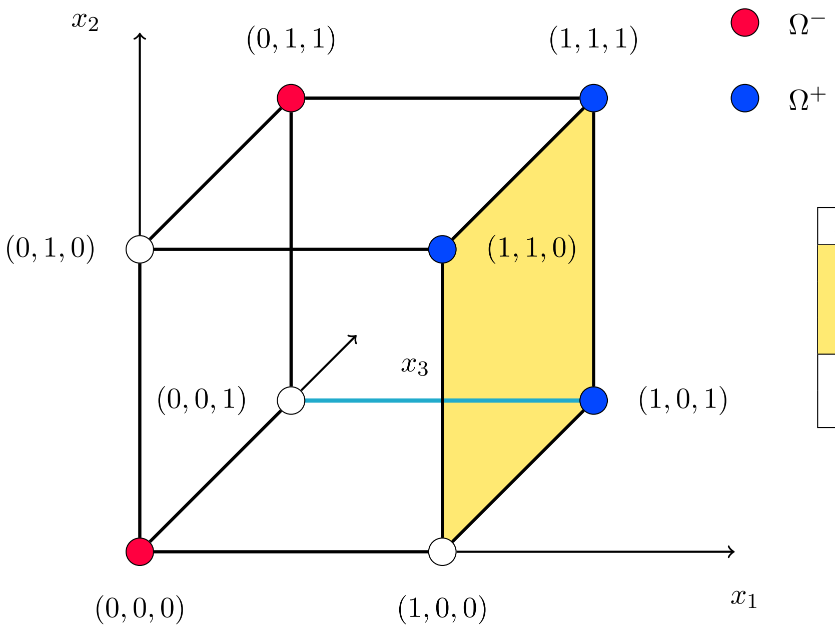
$$\Phi_f^+(x) = \begin{cases} 1, & \text{if } x \notin \Omega^-; \\ 0, & \text{if } x \in \Omega^-; \end{cases}$$

and

$$\Phi_f^-(x) = \begin{cases} 1, & \text{if } x \in \Omega^+; \\ 0, & \text{if } x \notin \Omega^+. \end{cases}$$

For any extension e of $f = (\Omega^+, \Omega^-)$ we get $\Phi_f^-(x) \leq e(x) \leq \Phi_f^+(x)$ for all $x \in \{0,1\}^n$. A major goal of LAD is to find a reasonable extension e for a given pdBf.

The main assumption standing behind the LAD methodology is that any real life data set does not consist of randomly distributed observations, but that there is a rational explanation for the data observed. We shall, therefore, use LAD to find some reasonable extension out of the 2^{2^n-k} ways to extend a pdBf that is defined on k points of the Boolean hypercube $\{0,1\}^n$. LAD provides concepts and methods to decide whether an extension is a *good* extension for a given pdBf. We will get an in-depth look at these concepts in the following sections.



x_1	x_2	x_3	
1	1	0	Ω^+
1	0	1	
1	1	1	
0	1	1	Ω^-
0	0	0	

Fig. 2.2.: A geometric intuition for patterns as subcubes of the n-dimensional hypercube.

2.3 Patterns

The key concept of LAD are patterns. This section is dedicated to the concept of patterns, their generation and application in LAD.

A pattern consists of literals. For a Boolean variable x, we denote its negation by $\overline{x} = 1 - x$. Both x and $\overline{x}$ are *literals*. A *term* is a conjunction of distinct literals. Every term t can be seen as a Boolean function. For a vector $v \in \{0,1\}^n$, we denote by $t(v)$ the binary value that results from applying the Boolean function t to v, where $t(v)$ is defined even if the degree of t is less than n. We say that a term t *covers* a point $v \in \{0,1\}^n$ if $t(v) = 1$.

Definition 2.1 (Positive (negative) pattern)**.** *Let* (Ω^+, Ω^-) *be a pdBf defined on a set* $\Omega = \Omega^+ \uplus \Omega^-$ *of observations. A term* t *is called a* positive (negative) pattern *of* (Ω^+, Ω^-) *if it covers at least one positive (negative) observation and no negative (positive) observation. The* degree *of a pattern is the number of its literals.*

In Figure 2.2 we see an example of a pdBf in three variables together with its geometric interpretation in the three-dimensional Boolean hypercube. Two positive patterns are marked in the data set and their geometric representation is shown. We note that a pattern of degree d of a pdBf in n variables is a $(n-d)$-dimensional subcube of the Boolean hypercube.

Positive and negative patterns are defined symmetrically. For reasons of clarity we will often restrict our statements to positive patterns. The same then holds analogously for negative patterns. The intuition of a positive pattern is that it shows a combination of variable values that has never appeared in a negative observation, but did appear for some of the positive observations. The fact that a new observation is covered by a positive pattern can be seen as an indication of its positive characteristic [24].

Note that every positive observation corresponds to a positive pattern, and every negative observation to a negative pattern. Those patterns are called *minterms* or *characteristic terms*. Given the data set in Table 2.2 (partitioned by x_0), the positive observation 1 provides the positive pattern $\overline{x_1}\overline{x_2}x_3\overline{x_4}x_5x_6$ and the negative observation 5 provides the negative pattern $\overline{x_1}\overline{x_2}\overline{x_3}x_4x_5\overline{x_6}$.

Other (arbitrary) examples of positive patterns include $\overline{x_1}x_6$, $x_3\overline{x_5}$ and $x_1x_3x_4$. For instance, the positive pattern $\overline{x_1}x_6$ covers the positive observations 1 and 3 while it evaluates to 0 for the remaining observations, particularly considering the three that are negative.

To put the definition of patterns in relation to the concept of Boolean functions note the following Lemma [34].

Lemma 2.1. *Let $f = (\Omega^+, \Omega^-)$ be a pdBf. A term t is a positive pattern of f if and only if t is an implicant of its extension Φ_f^+ that covers some point in Ω^+.*

In the last paragraphs, we familiarized ourselves with the definition of a pattern in LAD. Besides this general definition of patterns, several types of patterns have been introduced in the LAD literature. We will follow these definitions in the next sections. When we talk about patterns as defined above we will often refer to *general patterns* to distinguish them from the pattern types, which we investigate in Section 2.3.1.

2.3.1 Types of patterns

When we think about the whole set of patterns, we note that in general the number of patterns is huge compared to the size of the data set. To gain more clarity within this large amount of patterns it is not only convenient but necessary to define properties of patterns that allow us to rank them according to certain criteria.

Various properties and types of patterns have been studied and their relative efficiency has been analyzed [57]. For further definitions we introduce the basic terminology of the theory of partially ordered sets [2].

Given a set A we denote by $A \times A$ the set of all ordered pairs (x, y) such that $x \in A$ and $y \in A$.

Definition 2.2 (Binary relation and partial (pre-)order)**.** *A* binary relation *on A is a subset of $A \times A$. We say that two elements $x, y \in A$ are* comparable *with respect to a binary relation ρ if either $(x, y) \in \rho$ or $(y, x) \in \rho$, and incomparable otherwise.*

A binary relation ρ on A is called partial pre-order *(or* quasi-order*) if it is*

- reflexive*, i.e. $(x, x) \in \rho \;\; \forall x \in A$;*
- transitive*, i.e. $(x, y) \in \rho$ and $(y, z) \in \rho$ implies $(x, z) \in \rho \;\; \forall x, y, z \in A$.*

A partial pre-order is called partial order *if it is*

- antisymmetric*, i.e. $(x, y) \in \rho$ and $(y, x) \in \rho$ implies $x = y \;\; \forall x, y \in A$.*

We consider partial (pre-)orders on the set of patterns. Let (Ω^+, Ω^-) be a partially defined Boolean function of n variables and P a pattern of (Ω^+, Ω^-). We denote by

- $Lit(P)$ the set of literals in P;
- $S(P)$ the subcube of P, i.e. the set of points of $\{0, 1\}^n$ covered by P;
- $Cov(P)$ the set of true points of (Ω^+, Ω^-) covered by P, called the *coverage* of P.

These elaborations allow us to state the following preferences:

Definition 2.3 (Simplicity preference)**.** *A pattern P_1 is* simplicity-wise preferred *to a pattern P_2 if and only if the set of literals in P_1 is contained in the set of literals in P_2, i.e. $Lit(P_1) \subseteq Lit(P_2)$. We write $P_2 \leq_\sigma P_1$.*

Definition 2.4 (Selectivity preference)**.** *A pattern P_1 is* selectivity-wise preferred *to a pattern P_2 if and only if the subcube of pattern P_1 is included in the subcube of pattern P_2, i.e. $S(P_1) \subseteq S(P_2)$. We write $P_2 \leq_\Sigma P_1$.*

Definition 2.5 (Evidential preference)**.** *A pattern P_1 is* evidentially preferred *to a pattern P_2 if and only if the set of observations covered by P_1 includes the observations covered by P_2, i.e. $Cov(P_2) \subseteq Cov(P_1)$. We write $P_2 \leq_\epsilon P_1$.*

We denote by $P_1 \leq_\rho P_2$ that pattern P_2 is preferred to P_1 according to preference ρ. The simultaneous satisfaction of $P_1 \leq_\rho P_2$ and $P_2 \leq_\rho P_1$ is denoted by $P_1 \approx_\rho P_2$ for any preference ρ.

Definition 2.6 (Pareto-optimal)**.** *Given a preference ρ on the set of patterns, a pattern P_1 is called* Pareto-optimal *with respect to ρ if there is no pattern P_2 with $P_2 \neq P_1$ and $P_1 \leq_\rho P_2$.*

Note that the relations $P_1 \approx_\sigma P_2$ and $P_1 \approx_\Sigma P_2$ imply that $P_1 = P_2$. Therefore, the simplicity preference and the selectivity preference are partial orders. The relation $P_1 \approx_\epsilon P_2$ does usually *not* imply $P_1 = P_2$. The evidential preference is a partial pre-order. We further note that obviously $P_1 \leq_\sigma P_2$ if and only if $P_1 \geq_\Sigma P_2$.

The defined preferences lead to consequential definitions of pattern types.

Definition 2.7 (Prime pattern)**.** *A pattern that is Pareto-optimal with respect to the simplicity preference σ, i.e. a pattern with an inclusion-wise minimal set of literals, is called* prime.

In other words, a pattern is prime if the removal of any of its literals results in a term that is not a pattern.

For the data set in Table 2.2, the positive pattern $\overline{x_1}x_6$ is a prime pattern given that neither $\overline{x_1}$ nor x_6 is a positive pattern itself, because each of these terms cover one of the negative observations. On the other hand, $x_1x_3x_4$ is not prime because x_1 can be removed to obtain the smaller positive pattern x_3x_4, which is prime because the literals x_3 and x_4 each cover one of the negative observations.

Definition 2.8 (Strong pattern)**.** *A pattern that is Pareto-optimal with respect to the evidential preference ϵ is called* strong.

A pattern P_1 is strong if there is no pattern P_2 such that $Cov(P_1) \subset Cov(P_2)$. An example for a strong pattern in Table 2.2 is $\overline{x_1}x_6$. This pattern covers observation 1 and observation 3 and we can not find a pattern that covers a larger set of observations that includes observation 1 and 3.

The simplicity preference and the evidential preference have let us to the definitions of prime and strong patterns, respectively. We note that a definition of a pattern type depending on the selectivity preference does not provide us with a greater insight as the selectivity-wise preferred positive (negative) patterns are exactly the minterms of the positive (negative) observations. In the following we will see that the selectivity preference is, nevertheless, very useful when we combine it with one of the other preferences. In order to do so we define the following:

Definition 2.9 (Intersection and lexicographic refinement)**.** *Given two preferences π and ρ on the set of patterns:*

- *A pattern P_1 is preferred to a pattern P_2 with respect to the intersection $\phi \wedge \rho$ if and only if $P_2 \leq_\phi P_1$ and $P_2 \leq_\rho P_1$.*
- *A pattern P_1 is preferred to a pattern P_2 with respect to the lexicographic refinement $\phi|\rho$ if and only if either $P_2 <_\phi P_1$ or $P_1 \approx_\phi P_2$ and $P_2 \leq_\rho P_1$.*

Note that for any preference ϕ that is a partial order and for any preference ρ the lexicographic refinement $\phi|\rho$ is equal to ϕ.

Outgoing from the previous definitions we can define types of patterns according to combinations of preferences. Before doing so we take a look at the possible preference combinations and we will observe that out of the six possible combinations only three define new types of patterns [57].

We remind ourselves that the selectivity preference and the simplicity preference are contrary. Therefore, any combination between these two does not make any sense. Furthermore it can easily be seen that the following holds:

$$P_1 \geq_\sigma P_2 \Rightarrow P_1 \geq_\epsilon P_2.$$

Therefore, the evidential preference is in fact a refinement of the simplicity preference and the intersection $\sigma \wedge \epsilon$ equals σ. Having in mind that simplicity and selectivity are partial orders and that the lexicographic refinement $\phi|\rho$ is equal to ϕ for any partial order ϕ, we get that $\sigma|\epsilon = \sigma$ and $\Sigma|\epsilon = \Sigma$. Concluding from the previous paragraphs we get that only $\epsilon|\sigma$, $\epsilon|\Sigma$ and $\Sigma \wedge \epsilon$ lead to new preferences. We introduce the according pattern types in the following definitions

Definition 2.10 (Spanned pattern). *A pattern that is Pareto-optimal with respect to $\Sigma \wedge \epsilon$ is called* spanned.

Examples for spanned patterns in Table 2.2 are $\overline{x_1}x_2\overline{x_3}x_4x_5\overline{x_6}$ and $x_2x_3\overline{x_5}x_6$.

For the pattern types that are Pareto-optimal with respect to $\epsilon|\sigma$ and $\epsilon|\Sigma$ we do not need to introduce new names, because the following theorems hold [57]:

Theorem 2.1 (Strong prime). *A pattern is Pareto-optimal with respect to $\epsilon|\sigma$ if and only if it is both strong and prime.*

Theorem 2.2 (Strong spanned). *A pattern is Pareto-optimal with respect to $\epsilon|\Sigma$ if and only if it is both strong and spanned.*

Preference	Pareto-optimal pattern
σ	Prime
Σ	Minterm
ϵ	Strong
$\Sigma \cap \epsilon$	Spanned
$\epsilon \vert \sigma$	Strong prime
$\epsilon \vert \Sigma$	Strong spanned

Tab. 2.3.: Pareto-optimal pattern types in LAD.

An overview of the pattern types is given in Table 2.3.

The LAD methodology underlies a theoretical development and, therefore, the definition of some of the pattern types changed throughout the years. When LAD was first introduced in [35] general patterns were what we call prime patterns now. The basic definition of a pattern (nowadays prime pattern) is strongly related to the definition of *prime implicants* studied by W. Quine [86] in the 1950s, which are inclusion-wise minimal implicants of Boolean functions. We note that in the special case of Boolean functions, i.e. $\Omega^+ \cup \Omega^- = \{0, 1\}^n$ the definition of prime patterns and the definition of strong patterns actually coincide in that of prime implicants.

Theorem 2.3. *[35] Every prime pattern of a pdBf* (Ω^+, Ω^-) *is a prime implicant of its extension* $\Phi^+_{(\Omega^+, \Omega^-)}$.

Besides the pattern types introduced, which are based on the preferences discussed, other pattern types have been developed in the LAD literature. We focus on one more pattern type, the *maximal* patterns, in the following paragraphs as we will make use of it throughout the thesis.

Maximal patterns

Over the last decades various pattern types alongside the types defined by the preferences introduced by Hammer et al. [57] were studied. Most of them are based on the coverage of a pattern. In the next paragraphs and the following chapters we make further use of the definition of *maximal patterns*.

Definition 2.11 (Maximal pattern)**.** *A pattern is called* maximal *if it is maximal with respect to the number of observations covered.*

A pattern P_1 is maximal if there is no pattern P_2 such that $|Cov(P_1)| < |Cov(P_2)|$. Examples for maximal patterns in Table 2.2 are $\overline{x_1}x_3\overline{x_4}x_6$ and x_2x_4. All positive maximal patterns of the data set cover exactly two positive observations. We do not find a pattern covering more positive observations than that.

We want to put emphasis on the following rather easy remark to avoid confusions, as in literature maximal patterns are sometimes misleadingly referred to as strong patterns.

Remark 2.1. *A maximal pattern is a strong pattern, but a strong pattern need not be a maximal pattern.*

Proof. Let P_1 be a maximal pattern. Then $|Cov(P_2)| \leq |Cov(P_1)|$ for all patterns P_2. It follows directly that there exists no P_2 such that $Cov(P_1) \subset Cov(P_2)$. Therefore, P_1 is a strong pattern.

On the other hand let P_1 be a strong pattern. It holds that no pattern P_2 exists with $Cov(P_1) \subset Cov(P_2)$. This does not imply that there does not exist a pattern P' with a higher number of covered observations $|Cov(P_1)| < |Cov(P')|$, because we do not demand set inclusion here. For the construction of a counterexample we consider the data set given in Table 2.2. An example for a negative strong pattern P_1 is $\overline{x_2 x_3}$ that covers only observation 5. We notice that no pattern exists that covers observation 5 and any other of the two negative observations, which confirms that P_1 is a strong pattern as it covers an inclusion-wise maximal set of observations. However, negative patterns exist, which cover observation 6 and 7 simultaneously. An example for such a pattern is $x_1\overline{x_4}$. Therefore, P_1 is not a negative maximal pattern, because negative patterns having a coverage with higher cardinality exist. □

The definition of evidentially preferred patterns evolved over the last decades. We want to stick with the original definition, which we illustrated here. When evidentially preferred patterns were first introduced in [57], they were defined as patterns that have a maximal coverage with respect to set inclusion. In the article the authors argue that it is more useful to define strong patterns according to the property of having an inclusion-wise maximal coverage rather than having a maximal number of covered observations, because the coverage $Cov(P)$ of a pattern P can be seen as its "body of evidence". According to the authors it is, for that reason, a better relation as it takes the individual coverage into account and not just a number. In newer publications such as [93] maximal patterns are sometimes referred to as strong patterns. Here we do not follow this unfortunate choice of name.

In [7] the definition of maximal patterns, as given in Definition 2.11, was first presented. The authors of [22] refined the definition to the special case of *maximum*

α*-patterns*, which are patterns covering observation α while having a maximum coverage, i.e. a maximum number of covered observations of the correct sign. In [54] the authors defined C*-maximum patterns* for a subset $C \subseteq \Omega^+$ as patterns covering a maximum number of observations of the correct sign while also covering a set of reference observations. We see that the C-maximum patterns in fact are what we call strong patterns according to [57].

Is one pattern type better than another?

After the introduction of various types of preferences and patterns we want to facilitate the question of which pattern types are more useful than others. This question was addressed in various studies [57, 4, 7, 5]. We do not want to go into all the details here, but point out some of the results for a deeper understanding of patterns in LAD.

In [57] several arguments to build on patterns that are simplicity-wise preferred, namely prime patterns, are given. The first and most obvious one is that simple, or better, short patterns are easier to comprehend by humans. As a second argument the authors state that some studies showed [20] that simplicity of patterns leads to higher accuracy, although they point out that this result is not universally accepted [60]. In terms of LAD the use of the simplicity preference reduces the number of *false negatives*. However, it does not guarantee not to produce a lot of *false positives* as new observations are likely to be classified as positive when using short patterns only. To reduce false negatives, a natural way is to favor longer, i.e. *more selective* patterns.

In [5] the authors studied the problem of the use of *comprehensive*, i.e., long and selective, vs. *comprehensible*, i.e., short, patterns in LAD. Their study shows that selectivity-wise preferred patterns do provide a higher accuracy when used for classification with LAD but that the loss of accuracy using simple patterns is relatively small. The aspect of computational effort, therefore, is a reasonable point to think about when deciding for one of the two types for a certain application.

Summarizing, the question of which pattern type should be preferred can not be answered in general but should rather be discussed and weighed according to the data at hand.

2.3.2 Pattern parameters - Homogeneity and prevalence

The development of patterns in LAD can be seen as the development of a simple language for characterizing observations, which can easily be understood and communicated across application fields.

When we talk about applications apart from purely mathematical problems, one is always confronted with data sets that are not perfect in the sense of having one unique explanation. There might be errors coming from experiments or discretization. To balance these inaccuracies, parameters have been introduced in LAD [7], which allow some latitude in the definition of patterns. Two of them, which are most prominent and which we use in the sequel, are called *homogeneity* and *prevalence*.

Definition 2.12 (Homogeneity). *The* homogeneity $Hom^+(P)$ *of a positive pattern* P *is given by*

$$Hom^+(P) = \frac{|Cov(P)|}{|Cov_{total}(P)|}, \tag{2.1}$$

where $Cov(P)$ *is the set of positive observations covered by* P *and* $Cov_{total}(P)$ *is the set of observations covered by* P *in total. The homogeneity* $Hom^-(P)$ *of a negative pattern* P *is defined analogously.*

The homogeneity of a pattern puts the number of covered observations of positive sign in relation the the number of observations covered in total, in particular including the negative observations. A high homogeneity of a pattern means that it covers a lot more positive observations (or observations of the correct sign) than negative observations (or observations of the opposite sign).

Definition 2.13 (Prevalence). *The* prevalence $Prev^+(P)$ *of a positive pattern* P *is given by*

$$Prev^+(P) = \frac{|Cov(P)|}{|\Omega^+|}. \tag{2.2}$$

The prevalence $Prev^-(P)$ *of a negative pattern* P *is defined analogously.*

The prevalence of a pattern is a measure of how many positive observations are covered in relation to the number of positive observations in total. A positive pattern has a high positive prevalence if it covers a large amount of the positive observations.

In the recent literature on LAD, the parameters homogeneity and prevalence became more popular and with that standard measures when dealing with patterns. In [4] the authors name five basic parameters that are associated to each pattern, namely *sign*, *degree*, *type*, *homogeneity* and *prevalence*.

2.3.3 Algorithms for pattern generation

The generation of patterns is a central problem of LAD. Several algorithms for the calculation of patterns of different types have been developed. In this section, we introduce two of them. The first one is a term enumeration approach for the generation of prime patterns. The second one is a Mixed-Integer Linear Programming (MILP) approach for the calculation of maximal patterns.

Term enumeration approach for prime pattern generation

In the beginnings of LAD, a main focus lay on prime patterns. In [24] the authors introduced two opposing methods for the generation of prime patterns. A *top-down* and a *bottom-up* approach. The top-down method starts with the minterms of each positive (negative) observation and then systematically removes literals until arriving at a positive (negative) prime pattern. The bottom-up approach is shown in Algorithm 1. The algorithm generates prime patterns up to a given degree D. For that purpose it begins adding systematically one literal after the other checking at each stage whether the resulting term covers a positive observation and hence is a candidate term. In a second step it checks whether the candidate term does not cover any negative observations, which makes it a prime pattern. The term enumeration method is a costly process. For a binary data set with n variables $\sum_{i=1}^{n} 2^i \binom{n}{i}$ candidates have to be searched to enumerate all patterns up to degree $D = n$.

In the further development of LAD, Pareto-optimal patterns were introduced [57] and algorithms to transform patterns into Pareto-optimal ones. In [7] the authors introduced a consensus-type algorithm for the generation of spanned patterns called *SPIC - spanned patterns via input consensus*, which runs in polynomial time. Procedures for the calculation of all strong spanned and all strong prime patterns were proposed in [5] and a set covering algorithm for the generation of maximum patterns in [56]. An efficient algorithm for enumerating all prime patterns [9] and a branch-and-bound algorithm for patterns having maximum coverage [40] have been developed.

Algorithm 1 Term enumeration algorithm for the generation of prime patterns up to a given degree [24].

```
input:
    $B^+, B^- \subset \{0,1\}^n$          ▷ Sets of positive and negative Boolean points
    $D$                                   ▷ Maximal degree of patterns to be generated
output:
    $P$                                   ▷ Set of prime patterns of degree up to $D$
initialize:
    $P := \emptyset$;
    $C_0 := \{\emptyset\}$
for $d := 1$ to $D$ do
    if $d < D$ then
        $C_d := \emptyset$;
    end if
    for $T$ through $C_{d-1}$ do
        $p$ :=maximal index of literal in $T$;
        for $s := p + 1$ to $n$ do
            for $l_{new}$ through $\{l_s, \overline{l_s}\}$ do
                $T^{'} := T \wedge l_{new}$;
                for $i := 1$ to $d - 1$ do            ▷ All subterms in $C_{d-1}$?
                    $T^{''}$ := $T$ with $i$th literal dropped;
                    if $T^{''} \notin C_{d-1}$ then
                        go to $\diamond$;
                    end if
                end for
                if $1 \in T^{'}(B^+)$ then            ▷ Positive observation covered
                    if $1 \notin T^{'}(B^-)$ then     ▷ No negative observation covered
                        $P := P \cup \{T^{'}\}$;      ▷ $T^{'}$ is prime pattern
                    else if $d < D$ then              ▷ Negative observation covered
                        $C_d := C_d \cup \{T^{'}\}$;  ▷ $T^{'}$ is candidate
                    end if
                end if
                $\diamond$;
            end for
        end for
    end for
end for
```

$$
\begin{aligned}
\min_{z,y,d} \quad & \sum_{i \in \Omega^{\bullet}} y_i && (2.3) \\
\text{s.t.} \quad & \sum_{j=1}^{2n} a_{ij} z_j + n y_i \geq d, \quad i \in \Omega^{\bullet} && (2.4) \\
& \sum_{j=1}^{2n} a_{ij} z_j \leq d - 1, \quad i \in \Omega^{\bar{\bullet}} && (2.5) \\
& z_j + z_{n+j} \leq 1, \quad j \in N && (2.6) \\
& \sum_{j=1}^{2n} z_j = d && (2.7) \\
& \sum_{i \in \Omega^{\bullet}} y_i \leq |\Omega^{\bullet}| - 1 && (2.8) \\
& 1 \leq d \leq n && (2.9) \\
& z \in \{0,1\}^{2n} && (2.10) \\
& y \in \{0,1\}^{m^{\bullet}} && (2.11)
\end{aligned}
$$

Fig. 2.3.: MILP adapted from [93] for the generation of maximal patterns.

A major development in pattern generation are Mixed-Integer Linear Programs (MILP). In the following paragraphs we give an insight in how MILP can be utilized for pattern generation using the basic example of maximal patterns.

MILP approach for maximal pattern generation

A Mixed-Integer Linear Programming (MILP) approach for the generation of different types of patterns was proposed in [93] and further studied in [106, 107, 105].

In their first publication on the topic [93], the authors formulated a MILP generating maximal patterns and adjusted the program for the calculation of maximal prime and maximal spanned patterns. We follow this construction and explain here in more detail the MILP adapted from [93] for maximal pattern generation. It is shown in Figure 2.3.

The main idea of the program is to minimize the number of observations in the data set that are *not* covered by the generated pattern. This guarantees that the pattern we calculate is a maximal pattern by definition. We introduce a few notations according to [93], which are needed for understanding the MILP.

In the following let $\bullet \in \{+,-\}$ be a variable such that $\Omega^\bullet$ stands for either Ω^+, the class of positive observations, or Ω^-, the class of negative observations, depending on whether positive or negative patterns are calculated. We notate with $\bar{\bullet}$ the opposite of $\bullet$. Suppose there are $m^\bullet$ observations of type $\bullet$. Each of the observations is described by n attributes x_j, $j \in N = \{1,\ldots,n\}$. For each observation O_i, $i \in \Omega^\bullet$, $\bullet \in \{+,-\}$, let a_{ij} denote the binary value of the j-th attribute in observation O_i. For the negation of the variables x_j, $j \in N$ we introduce n additional attributes x_{n+j}. We denote with l_j the j-th literal, which is either $l_j = x_j$ or $l_j = x_{n+j}$. A term $t = \bigwedge_{j \in N'} l_j$, $N' \subseteq N$ is a conjunction of literals and d is the degree of the term. We call a term t a $\bullet$ pattern, $\bullet \in \{+,-\}$, if it is a positive or negative pattern, respectively. For a term t of degree d we use binary decision variables z_j, $j = 1,\ldots,2n$ to notate whether literal l_j is included in t or not. This means that l_j is included in the term if and only if $z_j = 1$, for each $j = 1,\ldots,2n$. For all $j \in N$ we have that $x_j + x_{n+j} = 1$. This implies that

$$z_j + z_{n+j} \leq 1, \quad \forall j \in N.$$

For a term of degree d, we get that

$$\sum_{j=1}^{2n} z_j = d.$$

We define a function $D(O_i, z) = \sum_{j=1}^{2n} a_{ij} z_j$. We consider a $\bullet$ pattern $z = (z_1,\ldots,z_{2n})$. This implies that the vector z differentiates at least one $\bullet$ observation from all $\bar{\bullet}$ observations. Therefore, $D(O_i, z) = \sum_{j=1}^{2n} a_{ij} z_j < d$ for all O_i, $i \in \Omega^{\bar{\bullet}}$ and at least one observation O_q, $q \in \Omega^\bullet$ exists for which $D(O_q, z) = \sum_{j=1}^{2n} a_{qj} z_j \geq d$. We note that we might also have $D(O_k, z) = \sum_{j=1}^{2n} a_{kj} z_j < d$ for some $k \in \Omega^\bullet$ as the pattern might not cover all the $\bullet$ observations.

We introduce $m^\bullet$ binary variables y_i for O_i, $i \in \Omega^\bullet$. We set $y_i = 0$ if $D(O_i, z) \geq d$, meaning that the observation O_i is covered by the pattern associated with the vector $z = (z_1,\ldots,z_{2n})$, and $y_i = 1$ if $D(O_i, z) < d$, meaning that the observation O_i is not covered by the pattern associated with the vector $z = (z_1,\ldots,z_{2n})$. Then it follows that

$$D(O_i, z) = \sum_{j=1}^{2n} a_{ij} z_j \leq d - 1, \quad \forall i \in \Omega^{\bar{\bullet}},$$

and

$$D(O_i, z) = \sum_{j=1}^{2n} a_{ij} z_j + n y_i \geq d, \quad \forall i \in \Omega^{\bullet}.$$

The sum $\sum_{i \in \Omega^{\bullet}} y_i$ equals the number of $\bullet$ observations that are not covered by a pattern

$$t = \bigwedge_{z_j + z_{n+j} = 1} l_j = \bigwedge_{z_j = 1} x_j \bigwedge_{z_{n+j}=1} \overline{x_j}, \quad j \in N.$$

Theorem 2.4. *Let (z, y, d) be a feasible solution of the MILP in Figure 2.3. Then P defined as*

$$P := \bigwedge_{z_j = 1, j \in N} x_j \bigwedge_{z_{n+j}=1, j \in N} \overline{x_j}$$

forms a $\bullet$ pattern whose degree is d.

Proof. We begin by showing that the MILP in Figure 2.3 has at least one feasible solution. For this purpose we select any observation O_q, $q \in \Omega^{\bullet}$ and set $z_j = 1$ for $a_{qj} = 1$ and $z_j = 0$ otherwise. Further we set $y_i = 1$ for $i \in \Omega^{\bullet}$, $i \neq q$. Then we get $\sum_{j=1}^{2n} z_j = n = d$ and $z_j + z_{n+j} = 1$ for all $j \in N$. The constructed solution (z, y, d), therefore, satisfies all constraints of the MILP and is a feasible solution.

Next we consider a feasible solution (z, y, d) of the MILP and define $N_t := \{j = 1, \dots, 2n \mid z_j = 1\}$. Based on (2.8) we can assume a $\bullet$ observation O_l, $l \in \Omega^{\bullet}$ for which $y_l = 0$ in the solution. Then from (2.4) and (2.7) follows that $a_{lj} = 1$ for all $j \in N_t$ for O_l and from (2.5) and (2.7) follows that $a_{kj} = 0$ for at least one $j \in N_t$ for each O_k, $k \in \Omega^{\overline{\bullet}}$. This yields for the term associated with the solution (z, y, d) applied on the observation O_l:

$$\prod_{j \in N_t} a_{lj} = \prod_{z_j = 1, j \in N} a_{lj} \prod_{z_{n+j}=1, j \in N} \overline{a_{lj}} = 1.$$

For all observations O_k, $k \in \Omega^{\overline{\bullet}}$ we get

$$\prod_{j \in N_t} a_{kj} = 0.$$

Therefore, the term associated with the solution covers at least one $\bullet$ observation, namely O_l by construction and none of the $\overline{\bullet}$ observations. Hence the term is a $\bullet$ pattern. The constraints given in (2.6) and (2.7) imply the degree $\sum_{j=1}^{2n} z_j = d = |N_t|$ of the pattern, where $|N_t|$ is the cardinality of the set N_t. □

Theorem 2.5. *Let (z, y, d) be an optimal feasible solution of the MILP in Figure 2.3. Then P is a maximal $\bullet$ pattern whose degree is d.*

Proof. Theorem 2.4 implies the existence of a feasible solution. The optimum of the MILP is bounded from below by 0. These two observations guarantee the existence of an optimal solution for the MILP.

Further, we recall that y_i, $i \in \Omega^\bullet$ having value 1 in a solution of the MILP indicates that the corresponding observation O_i is *not* covered by the pattern that is formed as described in Theorem 2.4 in association to the solution. The objective function of the MILP minimizes the sum over the y_i and, therefore, minimizes the number of observations that are not covered by the resulting pattern. As a result a pattern constructed from an optimal solution of the MILP given in Figure 2.3 has maximum coverage among all $\bullet$ patterns and is by definition a maximal pattern. □

In [93] the authors show how the MILP presented in Figure 2.3 can be modified to generate different types of patterns, namely maximal prime and maximal spanned, by only changing the objective function. They further introduce homogeneity and prevalence to the Mixed-Integer Linear Program.

Within this last section we studied the concept of patterns in LAD. We learned about various pattern types, their characteristics and advantages. In the next section we focus on how patterns can be combined to form different types of theories.

2.4 Theories

The theory formation is the final step in the data analysis with LAD (see Figure 2.1). As mentioned before, a major goal of LAD is to find a reasonable extension e for a given pdBf $f = (\Omega^+, \Omega^-)$. In this section, we introduce different classes of extensions, explain how predictions of new observations can be made in practice by the construction of a discriminant and give an insight in existing algorithms for theory formation.

For a pdBf $f = (\Omega^+, \Omega^-)$ the disjointness of the sets of positive and negative observations is a necessary and sufficient condition for the existence of an extension of f. The actual calculation might, however, be a hard problem, which has extensively been studied (see, for example, [26]).

Definition 2.14 (Theory). *Let $f = (\Omega^+, \Omega^-)$ be a pdBf on $\Omega = \Omega^+ \cup \Omega^- \subset \{0,1\}^n$. An extension $e\colon \{0,1\}^n \to \{0,1\}$ with $e(x) = f(x)$ for all $x \in \Omega$ of f which can be represented as a Boolean expression by the disjunction of a set of patterns of f is a* theory.

Theories in LAD are constructed to understand certain phenomena and to make predictions for new observations. While an important part of the methodology regarding theories is of course how to build them (see Section 2.4.2), we first have a look at different classes of extensions and how they can model different behavior of the data and the underlying systems.

2.4.1 Classes of extensions

Given a pdBf $f = (\Omega^+, \Omega^-)$ with $\Omega = \Omega^+ \cup \Omega^- \subseteq \{0,1\}^n$ the number of all feasible extensions of $f = (\Omega^+, \Omega^-)$ is in general very large depending on n and the number of not observed vectors. In this subsection, we talk about some of the different classes of Boolean functions that were studied in the context of LAD and help to narrow the search space down when more details are known about the underlying process. Although the search space becomes smaller the problem EXTENSION(C) is shown to be NP-hard [26] for most of the classes C.

Problem:	EXTENSION(C)
Input:	A pdBf $f = (\Omega^+, \Omega^-)$ with $\Omega = \Omega^+ \cup \Omega^- \subseteq \{0,1\}^n$
Output:	An extension $e \in C$ of f

We want to describe some of the classes of Boolean functions in more detail. For a more comprehensive overview and an introduction to more classes of extensions we refer the reader to [2, 26].

Definition 2.15 (Positive extension). *An extension e is called* positive *or* monotone *if $X \leq Y \Rightarrow e(X) \leq e(Y)$, where $X = (x_1, \ldots, x_n) \leq Y = (y_1, \ldots, y_n)$ means $x_i \leq y_i$ for $i = 1, \ldots, n$.*

A positive extension carries the information that each attribute either has a contributing effect to the outcome or it has no effect at all. Regarding the introductory example given in Section 2.1 this means that a positive extension describing the phenomenon of getting a stomach ache from the consumption of food items implies that food items can not have an inhibiting effect on the stomach ache, i.e. either a food item contributes to the disease or it has no effect at all.

An elementary conjunction is called *positive* if it does not contain any complemented variables. The following theorem is well known:

Theorem 2.6. *[35] A Boolean function is positive if and only if its prime implicants are positive.*

A result regarding the LAD method following from Theorem 2.6 is:

Theorem 2.7. *[35] A pdBf has a positive extension if and only if its prime patterns cover all the positive observations.*

An extension, which is not positive, but can be tranformed into a positive function by a change of variables is called *unate*:

Definition 2.16 (Unate extension)**.** *An extension e is called* unate *if for some subset V of variables, the change of variables*

$$x' = \begin{cases} \overline{x}, & \textit{if } x \in V; \\ x, & \textit{otherwise,} \end{cases}$$

transforms e into a positive function.

Such a change of variables can for example be made for an extension $e = \overline{x}y$ with $x' = \overline{x}$ but no such change is possible for an extension of the form $e' = \overline{x}y \vee xz$.

For the introductory example such a unate extension carries the information that each attribute has a constant effect on the stomach ache, meaning that the effect is either contributing or inhibiting but not both in different combinations.

Classes of theories

Besides such extensions which carry additional information about the characteristic of the data by their type, theories of small order are of special interest, because interpretation and further calculations are more easy the simpler the theory (or extension) is.

The *order* $O(t)$ of a conjunction t is the number of literals in t. The order of a DNF $t_1 \vee t_2 \vee \cdots \vee t_k$ for conjunctions $t_1, \ldots, t_k$ is the maximum of $O(t_i)$ for $i = 1, \ldots, k$. Every Boolean function can be expressed as a DNF. We define the order of a Boolean function as the order of its expression(s) of lowest order.

According to [35], we call a pattern *redundant* with respect to a theory if the Boolean function obtained by omitting it from the set of patterns is still a theory. Similarly we call a literal *redundant* with respect to a pattern if the term obtained by omitting it is still a pattern.

Definition 2.17 (Thrifty theory)**.** *A theory is called* thrifty *if it does not contain any redundancies in the patterns it includes as well as in the literals included in its patterns.*

It can easily be seen that a thrifty theory is formed out of prime patterns. In [35] the authors stated the theorem below, which follows directly from the construction of a thrifty theory:

Theorem 2.8. *[35] If there exists a theory of order k for a pdBf then there exists a thrifty theory of order at most k. In particular, there exists a lowest order theory that is thrifty.*

A special class of theories, which we mentioned already in the very beginning of this thesis, is the class of *bi-theories*. In [23] the authors lie special emphasis on these kind of theories as they seem to be the most *justifiable* theories because they are supported both by the negative and the positive observations. Let us explain this in more detail.

Following the definitions of Crama et al. [35] a *theory* (or sometimes called *positive theory*) is an extension of the pdBf (Ω^+, Ω^-) that can be written as a disjunction of positive patterns. A *co-theory* [23] (or sometimes called *negative theory*) is a theory for the pdBf (Ω^-, Ω^+) and thus a Boolean function in Disjunctive Normal Form built out of negative patterns.

Definition 2.18 (Bi-theory)**.** *A theory e is called a* bi-theory *if $\overline{e}$, the complement of e, is a co-theory.*

Bi-theories play a special role in the LAD methodology because they are not only based on the observations of positive patterns leading to a positive classification, but also have a supported result regarding the negative classification.

In [23] the authors state that the class of reasonable Decision Trees is a subset of the class of bi-theories. According to [23] we call a Decision Tree D *reasonable* for (Ω^+, Ω^-) if

1. D defines an extension of (Ω^+, Ω^-);
2. for every leaf u of D, at least one observation of (Ω^+, Ω^-) is classified into u;

3. for every nonterminal vertex v of D, at least one positive observation $O^+ \in \Omega^+$ is classified into a descendant of v, and at least negative observation $O^- \in \Omega^-$ is classified into another descendant of v.

Reasonable Decision Trees are a well-studied field [74, 87, 28].

2.4.2 A discriminant to make predictions

In the last paragraphs, we introduced the concept of extensions and theories in LAD and how different classes can be used to model different behaviors of the underlying data set. In this subsection, we show how a theory can be built in practice and used for prediction with help of a discriminant [24].

There are several steps in the theory formation process where decisions have to be made according to the data and application. The first step is to select a representative subset of patterns. On the one hand, this subset should be large enough to capture all features of the data set. On the other hand, it should not be too large, because it might become hard to understand and might lead to uncertain classifications. To ensure that positive observations are assigned positive by the theory and negative observations are assigned negative, the subset of patterns should be chosen such that every positive (negative) observation is covered by at least one positive (negative) pattern. For this reason, we refer to such a subset of patterns as a *pattern cover*. An observation is classified as positive (negative) if it is covered by some of the positive (negative) patterns in the theory and by no negative (positive) pattern.

In the case that an observation is covered by both positive and negative patterns of the pattern cover, we construct a discriminant to weigh the positive and negative characteristic of the observation. Therefore, relative weights are assigned to each of the patterns: The *discriminant* Δ for $v \in \{0,1\}^n$ is given by

$$\Delta(v) = \sum_k w_k^+ P_k^+(v) + \sum_l w_l^- P_l^-(v), \tag{2.12}$$

where $P_1^+, \dots, P_k^+$, $k \in \mathbb{N}$ are the positive patterns with their positive weights $w_1^+, \dots, w_k^+$ and $P_1^-, \dots, P_l^-$, $l \in \mathbb{N}$ are the negative patterns with negative weights $w_1^-, \dots, w_l^-$. For classification, a threshold t has to be chosen. A vector v is classified as positive if $\Delta(v) > t$ and negative if $\Delta(v) \leq t$.

To increase the separating power of the discriminant described above the authors in [24] proposed to require that each positive (negative) point has to be covered by several positive (negative) patterns. The weights for the patterns can be chosen according to the application. In [24] the authors proposed to weight the patterns according to their properties like low degree or high coverage. We notice again that the LAD method gives space for the adjustment of the model according to the application.

2.4.3 Algorithm for theory formation

In the next paragraph, we introduce an algorithm proposed by Hammer et al. [35] and taken up by Ryoo et al. [106] later for the calculation of a pattern cover.

An iterative algorithm for the generation of a pattern cover

As we described before, a main part of the theory formation step is the selection of a reasonable pattern cover for the given set of positive and negative observations. In the first paper on LAD [35] the authors proposed a procedure to calculate a pattern cover for a given data set. This algorithm was later adapted in [106] under the name `PattGen`. In Algorithm 2 we show the proposed procedure. The algorithm successively calculates a positive pattern covering some of the given positive observations. These positive observations covered are deleted in the next step from the set of positive observations for which a pattern that covers them still has to be chosen. This way a positive pattern cover is generated. As a final step the generated patterns have to be checked for redundancies, which might occur due to the construction of the algorithm. To get a pattern cover of all positive and negative observations, the same procedure has to be implemented for the negative observations analogously. Besides the data set the algorithm takes a pattern type T as input. Any type introduced in the former section can be chosen here, although one has to pay attention to the fact that it may not be possible to find a pattern cover consisting only of patterns of the given type. We will make use of the algorithm explained in the later chapters of this thesis.

At this point we have learned about the major steps *pattern generation* and *theory formation* of LAD. These are the only two steps forming the LAD methodology as it was proposed in [35]. However, we already mentioned that, since data is usually not binary, a preprocessing step had to be added to the LAD workflow. In the following section we explain how non-binary data is transformed into binary data in LAD as proposed by [24].

Algorithm 2 Algorithm for the generation of a positive pattern cover based on the procedure proposed in [35].

```
input:
    Data set $\Omega = \Omega^+ \cup \Omega^-$;
    Pattern type $T$
output:
    A set of positive patterns covering the whole set of positive observations

$O^+ := \Omega^+$;
while $O^+ \neq \emptyset$ do
    Calculate a pattern $P$ of type $T$ for $O^+$;
    $O^+ := O^+ \setminus Cov(P)$;
end while
```

2.5 Data binarization and preprocessing

LAD was originally designed for binary data sets and it has been shown that it works well for this case. Nevertheless, most real-life data sets are not binary. To make such data sets accessible to the method and, therefore, suitable as inputs for LAD it is a crucial step in the methodology to discretize them. Many methods for discretization have been developed and studied in the literature (see [66] for an overview). In this section, we describe how binarization is done according to [24]. This binarization workflow is still the state-of-the-art procedure in LAD [69].

The data binarization is subdivided into two steps, which are the *Introduction of Boolean variables* and the *Selection of a support set*. In fact, already the first step provides us with a binary representation of the original data set, but in general this binary data set is way too large for further calculations. For this reason, it is important to choose a support set of features that is sufficient to represent the original data and does not include any redundant information.

2.5.1 Introduction of Boolean variables

Non-binary variables can be of different type. The simplest type for our purpose are *nominal* (or *descriptive*) variables that are *not ordered*. An example for a nominal variable is “color”, whose values could be “red”,“blue” and “yellow”. To binarize such a variable x we introduce to each value v_s a Boolean variable $b(x, v_s)$ such that

$$b(x, v_s) = \begin{cases} 1, & \text{if } x = v_s; \\ 0, & \text{otherwise.} \end{cases}$$

Note that there is no need to introduce Boolean variables to nominal variables that take only two different values. In that case the values are simply renamed to “1” and “0”.

In the special case where the values of nominal variables can be *ordered* as in, for example, “blood pressure” taking the values “low”, “normal” and “high” we follow a different procedure, which we explain in the following part. Those ordered nominal variables will be treated the same way as numerical variables.

Examples for variables having *numerical* values are “temperature” and “weight”. As for the ordered nominal variables described before the values of numerical variables follow an ordering. In many real life scenarios numerical variables are binarized in practice. For the example of “body temperature” this could be the translation to “normal” or “abnormal” depending on whether the temperature is inside or outside a certain interval or, in other words, is below or exceeds a given threshold. The binarization of the variable works by comparing the values to a so called *cut-point*. The question of which cut-points should be used in LAD discretization has been studied in [12]. In the following we explain how cut-points are used to binarize numerical variables according to [24]. For each numerical variable (or ordered nominal variable) we introduce two families of Boolean variables.

The first Boolean variables associated to each cut-point are the *level variables*. They show whether the value of the original variable is above or below the cut-point. For every variable x and cut-point t we introduce the Boolean variable $b(x,t)$ such that

$$b(x,t) = \begin{cases} 1, & \text{if } x \geq t; \\ 0, & \text{if } x < t. \end{cases}$$

The second type of Boolean variables introduced are called *interval variables* and are associated with each pair of cut-points. They show whether the value of the original variable lies inside or outside the interval defined by the cut-points. For every variable x and pair of cut-points t' and t'' with $t' < t''$ we introduce the Boolean variable $b(x,t',t'')$ such that

$$b(x,t',t'') = \begin{cases} 1, & \text{if } t' \leq x < t''; \\ 0, & \text{otherwise.} \end{cases}$$

The number of observations in the data set is finite and, therefore, each ordered variable takes only a finite number of values. Let these values be $v_1 < v_2 < \cdots < v_q$. Two cut-points t' and t'' with $v_{s-1} < t'$ and $t'' \leq v_s$ produce the same Boolean

	x_1	x_2	x_3	
1	yellow	5	high	Ω^+
2	blue	1	low	
3	yellow	3	low	
4	red	2	high	
5	red	6	high	Ω^-
6	blue	3	high	
7	blue	2	low	

Fig. 2.4.: An example of a non-binary data set.

variables on this data set. Having this in mind we do not have to consider any cut-points that lie outside the interval $[v_1, v_2, \dots, v_q]$. We define the cut-point t_s by

$$t_s = \frac{1}{2}(v_{s-1} + v_s).$$

To reduce the number of cut-points that are considered, we introduce the following definition. We call a cut-point t_s *essential* if there exist both a positive and a negative observation in the data set such that one of them has $x = v_s$ while the other one has $x = v_{s-1}$. It is easy to see that it is sufficient to use essential cut-points only.

In Figure 2.4 we show an example of a non-binary data set. In Figure 2.5 the binarization of the example is shown via introduction of Boolean level and interval variables.

2.5.2 Selection of a support set

As mentioned in the beginning of this section, we can translate a given data set including non-binary variables to a binary data set by introducing Boolean variables according to suitable cut-points only. The binary data set produced this way is very large and usually carries a lot of redundant information. When discretizing data sets bigger than the one seen in the example in Figure 2.4 the number of Boolean variables introduced this way grows rapidly and makes further computations hard. To make the binarization step useful in practice we show, in this subsection, how the size of the binary data set can be reduced by the selection of a support set. The main property of the data sets we analyze using LAD is that they are partitioned into two disjoint subsets of positive and negative observations. This property, which we call *contradiction-free,* has to be maintained by any discretization of the data set. Therefore, we formulate the following definitions:

	b_1	b_2	b_3	b_4	b_5	b_6	b_7
	$x_1 =$ red	$x_1 =$ blue	$x_1 =$ yellow	$x_2 \geq 1,5$	$x_2 \geq 2,5$	$x_2 \geq 4$	$x_2 \geq 5,5$
1	0	0	1	1	1	1	0
2	0	1	0	0	0	0	0
3	0	0	1	1	1	0	0
4	1	0	0	1	0	0	0
5	1	0	0	1	1	1	1
6	0	1	0	1	1	0	0
7	0	1	0	1	0	0	0

	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}
	$x_3 =$ high	$1,5 \leq x_2 < 2,5$	$1,5 \leq x_2 < 4$	$1,5 \leq x_2 < 5,5$	$2,5 \leq x_2 < 4$	$2,5 \leq x_2 < 5,5$	$4 \leq x_2 < 5,5$
1	1	0	0	1	0	0	1
2	0	0	0	0	0	0	0
3	0	0	1	1	1	1	0
4	1	1	1	1	0	0	0
5	1	0	0	0	0	0	0
6	1	0	1	1	1	1	0
7	0	1	1	1	0	0	0

Fig. 2.5.: Binarization of the data set given in Figure 2.4 with the help of level and interval variables.

Definition 2.19 (Support set)**.** *A set of Boolean variables of a binarization of a non-binary data set is a* support set *if the data set obtained by the elimination of all the other variables remains contradiction-free.*

Definition 2.20 (Irredundant)**.** *A support set is* irredundant *if no proper subset of it is a support set.*

To illustrate the approach of how to choose a support set we look at the binarization of the positive observation 1 and the negative observation 5 in Figure 2.5. They differ in the Boolean variables b_1, b_3, b_7, b_{11} and b_{14}. To ensure that those two observations remain contradiction-free, at least one of the Boolean variables in which they differ has to appear in every support set.

Let us introduce for each Boolean variable b_i a decision variable y_i with $y_i = 1$ if b_i is included in a support set and $y_i = 0$ otherwise. The above example then gives us for every support set the constraint

$$y_1 + y_3 + y_7 + y_{11} + y_{14} \geq 1.$$

We generalize this approach. Let $b_1, \ldots, b_q$ be a set of Boolean variables and $y_1, \ldots, y_q$ the set of Boolean decision variables associated to them as above. For a positive observation p and a negative observation n we define by $I(p, n)$ the set of indices in

$$\begin{array}{c}(1,5)\\(1,6)\\(1,7)\\(2,5)\\(2,6)\\(2,7)\\(3,5)\\(3,6)\\(3,7)\\(4,5)\\(4,6)\\(4,7)\end{array}\begin{pmatrix}1&0&1&0&0&0&1&0&0&0&1&0&0&1\\0&1&1&0&0&1&0&0&0&1&0&1&1&1\\0&1&1&0&1&1&0&1&1&1&0&0&0&1\\1&0&1&0&0&0&1&0&0&0&1&0&0&1\\0&0&0&1&1&0&0&1&0&1&1&1&1&0\\0&0&0&1&0&0&0&0&1&1&1&0&0&0\\1&0&1&0&0&1&1&1&0&1&1&1&1&0\\0&1&1&0&0&0&0&1&0&0&0&0&0&0\\0&1&1&0&1&0&0&0&1&0&0&1&1&0\\0&0&0&0&1&1&1&0&1&1&1&0&0&0\\1&1&0&0&1&0&0&0&1&0&0&1&1&0\\1&1&0&0&0&0&0&1&0&0&0&0&0&0\end{pmatrix}\begin{pmatrix}y_1\\y_2\\y_3\\y_4\\y_5\\y_6\\y_7\\y_8\\y_9\\y_{10}\\y_{11}\\y_{12}\\y_{13}\\y_{14}\end{pmatrix}\geq\begin{pmatrix}1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\end{pmatrix}$$

Fig. 2.6.: System of inequalities for the support set calculation for the example shown in Figure 2.4.

which p and n differ. Then $(y_1, \ldots, y_q)$ is the characteristic vector of a support set if and only if the following constraints are satisfied:

$$\sum_{i \in I(p,n)} y_i \geq 1 \qquad\qquad \forall p \in \Omega^+, \forall n \in \Omega^-. \tag{2.13}$$

A smallest support set can be found by solving the optimization problem that emerges from this:

$$\begin{aligned} \min \quad & \sum_{i=1}^{q} y_i \\ \text{s.t.} \quad & \sum_{i \in I(p,n)} y_i \geq 1 \quad \forall p \in \Omega^+, \forall n \in \Omega^- \\ & y_i \in \{0,1\}, \quad i = 1, \ldots, q. \end{aligned} \tag{2.14}$$

Here the selection of a support set is modeled as a set covering problem. In the case that the original data set involves monotone variables, we have to strengthen some of the constraints. This is due to the fact that we aim to choose a support set in such a way that the resulting pdBf has an extension in which all level variables corresponding to originally positive (negative) attributes remain positive (negative).

To do so, we eliminate those indices i in the set $I(p,n)$ for which the binary variable b_i is positive and $p_i = 0$ and $n_i = 1$; and those for which b_i is negative and $p_i = 1$ and $n_i = 0$. In Figure 2.6 the set of inequalities according to (2.13) for Example 2.4 is shown. With the help of (Mixed-) Integer Programming solvers it is possible to find an optimal solution to the above problem even for large data sets. Interestingly, the authors in [24] stated that for our purpose of finding a binarization for further calculations as pattern generation and theory formation, it is as good as (or even better) to find an approximate solution to (2.14). This is due to the goal that we want to achieve. On the one hand, features have to be selected to make the binary data set smaller and, therefore, reduce the complexity for the computations of patterns and theories. On the other hand, LAD aims to keep as much information as possible and make it available to the user. The process of selecting a support set becomes an act of balance between minimizing the number of features and not losing important information that we are not yet aware of. The support set selection process remains a field of research. In [30] the authors reformulated the problem using weighted set covering. For this purpose they propose to modify the constraints of (2.14) by adding coefficients and then use a greedy approach to solve the problem approximately. We explain this idea in the following.

Distinguishing true and false observations in more than one attribute

In order to make sure that true and false observations differ in more than one attribute in the resulting binarization, the right-hand side of the inequalities in (2.14) can be replaced by a higher value μ. The value of μ needs to be chosen depending on the application at hand and the size and structure of the given data set. While a natural bound on μ is that it can not exceed the Hamming distance between the set of positive and negative observations, another goal should be to keep the support set small and computationally feasible.

Give weight to the attributes according to their discriminating power

As a second step, weights can be assigned to the attributes y_i in the objective function in (2.14) to give preference to those having more discriminating power. Possible ways to choose these weights named in [24] are statistical measures like entropy or variance.

Give weight to the cut-points according to their discriminating power

The next modification is to assign weights as coefficients in the inequalities. This way cut-points having high discriminating power are favored. One possibility is to take the coefficient equal to the minimum of the two distances from the cut-point to the attribute values of the corresponding positive and negative observation, normalized by the spread of the corresponding attribute. The spread is the difference between the highest value of the attribute and the lowest value of the attribute in the data.

The resulting modifications of (2.14) are shown in (2.15).

$$\begin{aligned} \min \quad & \sum_{i=1}^{q} u_i y_i \\ \text{s.t.} \quad & \sum_{i \in I(n,p)} c_{ji} y_i \geq \mu_j, \quad j = 1, \ldots, m \\ & y_i \in \{0,1\}, \quad i = 1, \ldots, q. \end{aligned} \tag{2.15}$$

To solve the optimization problem shown in (2.15) the authors in [24] propose a greedy method. The procedure is shown in Algorithm 3.

Algorithm 3 Greedy algorithm for the solution of (2.15) [24].

```
input:
    $\{u_i\}$              ▷ $q$-vector of objective function coefficients
    $\{c_{ji}\}$           ▷ $m \times q$-matrix of constraint coefficients
    $\{\mu_j\}$            ▷ $m$-vector of right hand side coefficients
output:
    $\{y_i\}$              ▷ characteristic $q$-vector of support set
initialize:
    $y := 0$;
    $s \in \mathbb{R}^m; s := 0$;
while $s \ngeq \mu$ do
    choose $i^* \in \{i = 1, \ldots, q : y_i = 0\}$
    to maximize $\frac{1}{u_i} \sum_{j:\mu_j - s_j > 0} min\{1, \frac{c_{ji}}{\mu_j - s_j}\}$;
    $y_{i^*} := 1$;
    $s_j := s_j + c_{ji^*}, j = 1, \ldots, m$;
end while
```

The output of the greedy algorithm in Algorithm 3 is a set of support variables. By eliminating all the other variables from the binarized data set we obtain a smaller set of binary variables, which maintains the important properties and information of the data.

Besides the pure binarization of a data set other preprocessing steps can be considered. An important one described in [24] is how to deal with errors. We describe this in the following subsection.

2.5.3 Dealing with errors in data sets

Most of the real-life data sets are not perfect. Measurement errors, missing values or wrong classifications might occur. To cope with such imperfectness we shall follow certain steps [24] while preprocessing a data set, depending on the type of error.

Classification errors

There are two types of classification error, which manifest themselves by different behaviors during the pattern generation process. The first type of a classification error is that a negative observation is labeled as positive. We become attentive to this kind of error when positive patterns of low degree do not cover this observation assuming that the majority of observations are correctly classified. We shall follow two different approaches depending on whether there exist only a few such erroneously classified observations or there exist a lot of them. In the first situation, we remove the observations from the data set. In the second situation we generate some patterns of higher degree to cover the observations to make sure that these, apparently significant, observations are not left out of our consideration.

The second type of a classification error is that a positive observation is labeled as negative. In that situation we see that patterns covering a high number of positive observations also cover some isolated negative points. Those observations should then be disregarded in the evaluation of the patterns. In [24] the authors proposed to call a term a positive pattern if the ratio of negative and positive points covered by it does not exceed

$$\tau\frac{|\Omega^-|}{|\Omega^+|},$$

where τ is a problem dependent threshold. The authors state that in their numerical experiments $0 \leq \tau \leq 0,2$ led to the best results.

Missing attribute values

Another type of error that might occur in a data set is that of missing attribute values. This type of error is more striking than the type described before as it propagates through the whole procedure. When the value of an original attribute is missing then the values of the introduced Boolean variables are missing. If we want to construct a support set from a binary data set we cannot use variables with missing attributes in order to be able to distinguish between positive and negative observations.

The authors in [24] introduce a new definition of covering an observation in the case of a data set having missing attribute values.

Definition 2.21 (Weak covering and strong covering)**.** *A term covers an observation* weakly *if there exists an assignment of 0-1 values to the missing values in the observation such that the resulting 0-1 observation is covered by the term. A term covers an observation* strongly *if for any assignment of 0-1 values to the missing values in the observation the resulting 0-1 observation is covered by the term.*

Building up on this refinement of the covering definition, we can define *robust positive (negative) patterns* for data sets including missing attribute values.

Definition 2.22 (Robust patterns)**.** *A term is a* robust positive (negative) pattern *if it covers strongly at least one positive (negative) observation and it does not even weakly cover any negative (positive) observation.*

In the case of data sets without missing values the above definition reduces to the original definition of patterns. For further studies of the application of LAD to data sets with missing values see [25] and [27].

Measurement errors

The last type of error we refer to are measurement errors. Whenever data is produced by experiments or calculations, measurement errors occur. While the error might be small in the numerical case the impact of the error increases in the binarization process.

If the numerical value of an attribute is close to the chosen cut-point the value of the binarized variable can depend strongly on the error. In order to avoid such error-dependent variables the authors propose to consider a variable value as missing if it is close to the corresponding cut-point. For every attribute x and cut-point t we introduce a level variable

$$b(x,t) = \begin{cases} 1, & \text{if } x \geq t + \sigma_x; \\ 0, & \text{if } x < t - \sigma_x; \\ *, & \text{if } \sigma_x - t \leq x < t + \sigma_x, \end{cases}$$

where σ_x is an attribute-dependent measure of accuracy and $*$ stands for a missing attribute value. A similar modification has to be done for the interval variables.

When using the Boolean variables as described above we might introduce missing attribute values to data sets, which originally did not have any. The resulting problem has to be handled as described in the previous subsection.

2.6 Summary and discussion

In this chapter, we introduced the methodology of LAD. LAD is a method for machine learning and data analysis that is based on the generation and interpretations of patterns of various types.

We followed the three main steps of the method, namely *data binarization*, *pattern generation* and *theory formation*. While LAD can keep up with other known machine learning approaches like Support Vector Machines or Decision Trees regarding their prediction accuracy [24], it is especially interesting with respect to applications. The concept of patterns as compact pieces of information helps the communication of relevant data information across disciplines. For that reason, LAD is of particular interest in the context of biomedical research. In the following chapters we will show how patterns and theories based on the LAD method can be used in practice to analyze data sets from the field of life sciences.

Part II

Practice

3

AnswerSetLAD - A software package for LAD using Answer Set Programming

In the following chapter, we present the development of our software package AnswerSetLAD, which is available on GitHub [14]. This package was created on the basis of the LAD methodology. It is designed to automate the process of data analysis focusing on the generation of different kinds of patterns.
The idea arose in collaboration with Alexander Bockmayr, who recognized the usefulness of the LAD method in our interdisciplinary work field. I designed and implemented the preliminary version of an ASP model for pattern generation. In consultation with Martin Gebser and Torsten Schaub from the University of Potsdam the foundation of AnswerSetLAD was laid. We published a first conception of the design of a LAD software using an ASP framework [15]. This publication introduces the implementation of general and prime pattern generation, and the formation of minimal pattern covers only. In this chapter of the thesis, we describe our ASP extension of the software functionalities to all prominent LAD pattern types, as well as python tools for the binarization and theory formation step. Two further ASP approaches for theory formation are discussed in Chapter 4 under the aspect of our theoretical advancement of the LAD methodology.

In this chapter, we first introduce the requirements and goals for the development of the software package. We then illustrate why Answer Set Programming (ASP) is a suitable environment for the given tasks and give an introduction to the ASP language. Next, we explain the structure of AnswerSetLAD following the three work steps of LAD, namely *data binarization*, *pattern generation* and *theory formation*. The implementation of the LAD functionalities using ASP is described in detail. At the end of this chapter, we show in a performance study that for certain pattern types our ASP approach is able to solve problem instances much faster than the state-of-the-art Mixed-Integer Linear Programming (MILP) approach, which makes AnswerSetLAD a promising alternative within the LAD software available.

3.1 Goals and requirements

Within the previous chapters we got an insight into the methodology of LAD. This method, which combines ideas from Boolean functions, machine learning, combinatorics and optimization, provides a broad theoretical background for data analysis based on patterns. We discussed that LAD yields excellent results regarding prediction accuracy of classical data sets used for benchmarking in machine learning and is, therefore, a valid approach for classification problems. Especially regarding applications in interdisciplinary work fields, where results have to be communicated across different professional backgrounds, LAD is a beneficial method. LAD provides the user with comprehensible pieces of information in form of patterns. Particularly in the biomedical area LAD enjoys growing popularity [56].

In terms of application, any method is useless without a good software that integrates the theory. Over the past decades several software tools for LAD have been developed. These include a `C++` tool designed by Mayoraz [77], the `LAD-WEKA` software by Bonates and Gomes [21] written in `WEKA`, which is a data mining software package in `Java`, and `ladoscope` by Lemaire [70] in `OCaml`. Most of these tools are no longer maintained and although some of them like `ladoscope` offer a wide range of functions, new functionalities of LAD might be of interest, which are hard to add to the existing code.

Our main goal was to provide a software that includes the broad range of LAD functionalities and makes them accessible to the user. Of course, for the sake of utility, we wanted to use a framework that allows a good performance regarding running times on large data sets. Besides that the encoding should be as clear and succinct as possible such that, on the one hand, maintenance and integration of further features can be handled in an uncomplicated manner and, on the other hand, results can be understood and interpreted easily. We think that this is an obvious course of action as it goes in line with the original idea of LAD to make information communicable.

To achieve these goals, the programming paradigm Answer Set Programming (ASP) is a natural choice. The structure of the problem, namely the enumeration of patterns of Boolean expressions, is another important reason for choosing the ASP framework. ASP is in particular suitable for combinatorial (optimization) problems. We discuss ASP and its advantages in more detail in the following section.

3.2 Design

In the previous paragraphs we illustrated our goals for the software and explained our choice of the ASP framework on a general level. In this section, we want to go in more detail by introducing its syntax and semantics. We then give an overview of the structure of our software package `AnswerSetLAD` [14].

3.2.1 Answer Set Programming (ASP)

Answer Set Programming (ASP) [73, 13, 50] is a declarative programming paradigm. In contrast to imperative programming, the task for the user is to give a detailed description of *what the problem is* rather than explaining *how the problem should be solved*. The implementation of a problem in ASP consists thus of a concise representation of the problem by logical rules, which are then instantiated by an ASP grounder and solved by an ASP solver. ASP is particularly applicable for combinatorial (optimization) problems like the task of pattern generation in LAD. For that reason it is an eminently suitable choice.

The history of ASP

ASP is oriented towards difficult, primarily NP-hard, search problems. These search problems are reduced to the computation of so-called stable models or answer sets. The ASP methodology was first applied in 1997 by Dimopoulos et al. [36] and Soininen, Niemelä and Simons in 1998 [80]. One year later, answer set solvers and their application for difficult search problems were identified as a new programming paradigm by Marek and Truszczyǹski [76] and Niemelä [81].

Various systems for grounding and solving were developed since then (see for example [82, 49, 18]). For all our calculations we use the ASP system `clingo`, a combination of the grounder `gringo` and the solver `clasp` [49] developed by Potassco, the Potsdam Answer Set Solving Collection [33].

In the following section we give an introduction into the syntax and semantics of ASP. Our explanations are based on the language of `clingo`. A detailed description of its language and precise semantics can be found in [47].

Syntax and semantics

ASP is based on the stable model semantics of logic programming [72, 51]. Search problems are reduced to the computation of stable models, which are found by ASP solvers. Problems are formulated as logic programs that are finite sets of rules. A *rule* r is of the form

$$A_0 \text{ :- } A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_n, \tag{3.1}$$

where $n \geq m \geq 0$, each A_i, $0 \leq i \leq n$, is an *atom* and 'not' stands for negation by default. The left side of the rule is called *head* and the right side of the rule is called *body*. In the following, we explain the input language used by ASP systems like `clingo` [49]. A rule, as in (3.1), is a conditional constraint, meaning that the head must be true if the body is true. If $n = 0$, rule (3.1) is called a *fact* and denoted by

$$A_0.$$

Such a fact expresses that the atom A_0 is always true. Omitting A_0 in (3.1) amounts to taking A_0 to be false, and rule (3.1) represents an *integrity constraint*. Accordingly, the resulting rule

$$\text{:- } A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_n.$$

expresses that a stable model must not satisfy the body. Integrity constraints are thus often used to eliminate model candidates of a program.

To facilitate the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include *conditional literals* and *cardinality constraints* [96]. Conditional literals are of the form $A : B_1, \dots, B_m$, where A and B_i are possibly default negated literals for $0 \leq i \leq m$. Such conditional literals can be used to formulate cardinality constraints, which can be written as $s\ \{C_1; \dots; C_n\}\ t$, where each C_j is a conditional literal. The numbers $s, t \in \mathbb{N}$ provide lower and upper bounds on the number of satisfied literals in the constraint. The practical value of both constructs becomes apparent when used in conjunction with variables. For instance, a conditional literal like $a(X) : b(X)$ in a rule's body expands to the conjunction of all instances of $a(X)$ for which the corresponding instance of $b(X)$ holds. Similarly, $s\ \{a(X) : b(X)\}\ t$ holds whenever the number of true instances of $a(X)$ (subject to $b(X)$) is between s and t, $s, t \in \mathbb{N}$.

In addition to cardinality constraints, the input language of ASP provides further aggregates such as #min, #max and #sum. Optimization can be done using the ASP language. Objective functions minimizing the sum of weights w_i of literals B_i are expressed as `#minimize` $\{w_1 : B_1; \ldots ; w_n : B_n\}$. Specifically, we rely in the sequel on the input language of the ASP system `clingo` [49], as detailed in the corresponding user's guide [48].

The first step in the process of finding a solution to a problem is the grounding (e.g. by `gringo`) of rules that include variables, meaning that those variables are replaced by constants in ground instances. The grounded program is then passed to the solver (e.g. `clasp`), which computes the stable models of the program.

Definition 3.1 (Reduct)**.** *The* reduct P^X *of a program* P *relative to a set* X *of atoms is defined by*

$$P^X = \{head(r) \leftarrow body^+(r) \mid r \in P \text{ and } body^-(r) \cap X = \emptyset\},$$

where $body^+(r)$ *is the set of all positive atoms of the body and* $body^-(r)$ *is the set of all negative atoms of the body.*

Definition 3.2 (Stable Model)**.** *A set* X *of atoms is a* stable model *of a program* P *if the inclusion-wise minimal model of the reduct* P^X *of* P *relative to* X *is equal to* X.

For further questions on the ASP syntax and semantics we refer the reader to [48] for a detailed description of the `clingo` system and its use.

3.2.2 The software structure

Like the LAD methodology itself our software `AnswerSetLAD` can be divided into three main parts, namely *data binarization*, *pattern generation* and *theory formation* (Figure 3.1). While these can be seen as consecutive analyzing steps, they can be used individually depending on the result the user is interested in.

The first part of the software contains the features needed for data preprocessing in the case that the data set to be analyzed is not binary. The original LAD method is built on binary data sets. The procedures included in this part of the software follow very closely the proposed algorithms by Hammer et al. [24]. All methods belonging to this software part are implemented using `python`.

The second part includes all functionalities for the generation of patterns of different types. The pattern generation part contains, besides the enumeration of patterns

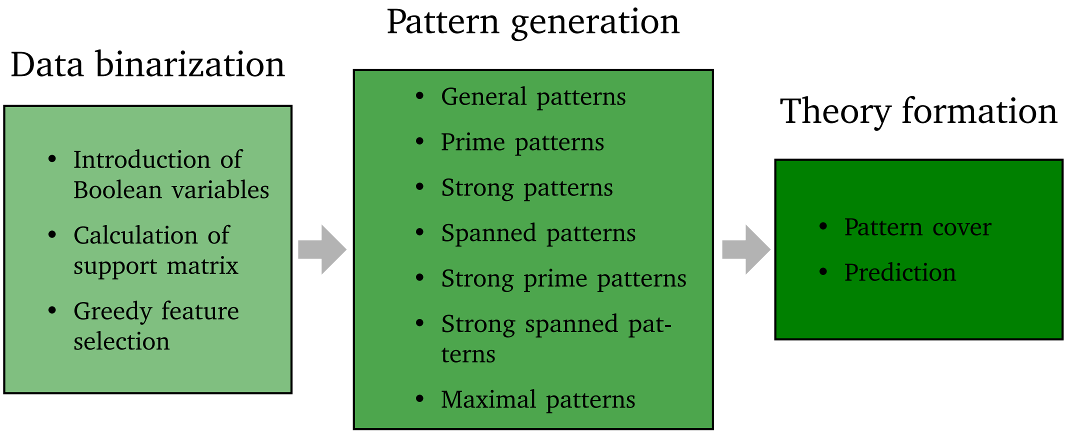


Fig. 3.1.: Overview of the structure and features of our software `AnswerSetLAD`.

without specific preference, programs to calculate *prime*, *strong*, *spanned*, *strong prime*, *strong spanned* and *maximal* patterns. All of the programs for pattern generation are implemented using ASP. As patterns play a key role in the LAD methodology, we invested most time and effort to work on this part of the software. Therefore, this work step of `AnswerSetLAD` is a central component of our software.

Part three of `AnswerSetLAD` contains several programs to form theories out of a set of generated patterns. This work step provides algorithms implemented in `python` as well as in ASP. Aside from algorithms described in former studies [35, 106], this part also includes programs based on our own theoretical extensions of LAD, which we discuss in more detail in Chapter 4.

3.3 Implementation

In this section, we describe the implementation of `AnswerSetLAD` [14]. It is organized in the three subsections *data binarization and preprocessing*, *pattern generation* and *theory formation* reflecting the analyzing steps of the LAD methodology and the structure of our software package.

3.3.1 Data binarization and preprocessing

The original LAD methodology is based on binary data sets. Investigated data is mostly not binary. This is the reason why the binarization step is a main part of the analysis. In this subsection, we explain how our toolbox `AnswerSetLAD` can be used to discretize and preprocess numerical data sets.

Running example data sets

We show the application of the binarization step on the example of six data sets from the UC Irvine Machine Learning Repository [38], which are widely used in the field of classification research and especially in the LAD literature. These data sets are *Breast Cancer Wisconsin (Original)* (BCW) [89, 75], *Heart Disease* (HD) [90], *BUPA Liver Disorders* (BLD), *Credit Approval* (CRED), *Pima Indians Diabetes* (PID) and *Boston housing* (HOUS). They will be used throughout this whole thesis.

In Table 3.1 the basic information on the six data sets are captured. The first column shows the number of observations times the number of attributes, which might be binary, numerical or nominal, after preprocessing the source data by deleting all observations that included missing values or appeared more than once. In more detail, for all attribute vectors that appeared more than once, belonging to the same class or to different classes, we kept only the first appearance of the vector and removed all the following. This leads to a disjoint data set.

The second column shows the number of level variables that were introduced while binarizing the attributes. These level variables are constructed as described in Section 2.5 and show whether the attribute value of the observation is above or below a certain threshold. Other than described in the methodology section and proposed by Hammer et al. [24], we use only level variables and did not introduce interval variables. With this decision we follow the proposal of Ryoo et al., who showed in [93] that it leads to approximately the same results in classification accuracy when leaving the interval variables out. At the same time this saves a lot of computational cost.

The last column of the table gives the number of observations times the number of selected 0-1 features after the greedy feature selection process. Again the observations are checked for disjointness, which leads to the smaller number of observations compared to the original data sets. The μ-value used for the greedy feature selection is shown in the third column. In the next paragraph, we discuss the workflow in detail.

Binarization workflow

The discretization procedure of `AnswerSetLAD` follows very closely the steps introduced by Hammer et al. [24] and described in Section 2.5. In Figure 3.2 the workflow of the binarization step is visualized.

	Original Observations^a × attributes	Number of level variables	μ	Binary Observations^b × selected features
BCW	449 × 9	72	6	161 × 20
HD	297 × 13	305	4	73 × 11
BLD	341 × 6	269	5	341 × 15
CRED	653 × 15	773	6	114 × 19
PID	768 × 8	857	5	28 × 24
HOUS	506 × 13	1217	6	14 × 20

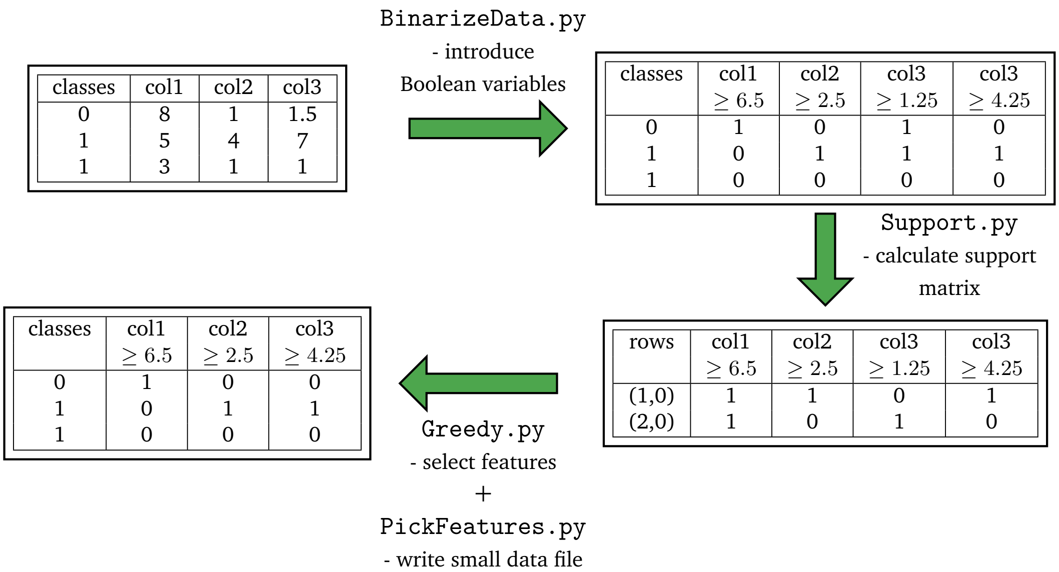
Tab. 3.1.: Basic information on the six data sets *Breast Cancer Wisconsin (Original)* (BCW), *Heart Disease* (HD), *BUPA Liver Disorders* (BLD), *Credit Approval* (CRED), *Pima Indians Diabetes* (PID) and *Boston housing* (HOUS) taken from the UC Irvine Machine Learning Repository [38]. [a] Disjoint data set after deletion of incomplete observations. [b] Disjoint data set.

The process starts with an input data file. This input file has to be in `csv`-format, where each row of the file represents one observation and each column represents one attribute. The first column called *classes* includes the class that the observation belongs to (0 or 1). In the following columns the attributes are listed under the names *col1, col2, . . . , colN*.

The `python` script `Binarize.py` is used to introduce the threshold variables, namely level variables and interval variables, if wanted, for each attribute. For the running data sets we used level variables only. The output of the script is a full binarization of the data set. In practice, it is not very useful though as it includes a lot of 0-1 variables that carry redundant information and the discretized data gets too big for pattern or theory calculations. For this reason, some of the variables have to be selected according to their usefulness to gain a smaller binary representation.

The next step in this direction is to calculate the support matrix consisting of the *difference vectors* of positive and negative observations. An entry of the difference vector for a positive and a negative observation equals 1 if the positive and the negative observation differ in this attribute and equals 0 otherwise. This is done using the program `Support.py`. The support matrix can then be used to ensure that the main property of the data, the disjointness of the positive and the negative class, stays intact. The number of rows of these support matrices is usually very large. The support matrix of the *BCW* data set consists of 50268 rows, for example.

To select features based on the calculated support matrix the python program `Greedy.py` is used. Because of the size of the support matrices we need to use an approximate solution to the feature selection problem and use the greedy heuristic proposed by Hammer et al. [24]. Other techniques for feature selection can be found in [94].



classes	col1	col2	col3
0	8	1	1.5
1	5	4	7
1	3	1	1

classes	col1 ≥ 6.5	col2 ≥ 2.5	col3 ≥ 1.25	col3 ≥ 4.25
0	1	0	1	0
1	0	1	1	1
1	0	0	0	0

rows	col1 ≥ 6.5	col2 ≥ 2.5	col3 ≥ 1.25	col3 ≥ 4.25
(1,0)	1	1	0	1
(2,0)	1	0	1	0

classes	col1 ≥ 6.5	col2 ≥ 2.5	col3 ≥ 4.25
0	1	0	0
1	0	1	1
1	0	0	0

Fig. 3.2.: Binarization workflow in our software package *AnswerSetLAD*.

Some parameters can be adjusted within the program to fit the underlying data set in the greedy procedure (see Chapter 2 Algorithm 3). These parameters are:

- u_i, $i = 1, \ldots, q$, the q-vector of the objective function coefficients;
- $c_{j,i}$, $j = 1, \ldots, m$, $i = 1, \ldots, q$, the $m \times q$- matrix of constraint coefficients;
- μ_j, $j = 1, \ldots, m$, the m-vector of the right-hand side coefficients with $1 \leq \mu \leq Hamm(\Omega^+, \Omega^-)$, where $Hamm(\Omega^+, \Omega^-)$ is the Hamming distance between the set of positive and the set of negative observations.

For all our calculations we used the variances of the attributes for the vector u, namely the variance of attribute i as entry u_i. This is one option proposed in [24]. Alternatively the authors named other statistical measures like entropy or Fisher's exact test statistic.

In our experiments, the entry $c_{j,i}$ of the constraint coefficient matrix c is, again according to [24], taken to be the minimum of the distance of the cut-point used in column i to the attribute values of p and p', the two observations belonging to row j, divided by the spread of the original attribute.

The choice of the parameter μ should be chosen according to the underlying data set. A μ value higher than 1 ensures that every positive observation differs in more than one attribute from every negative observation. Therefore, a higher value for μ is preferable. On the other hand, higher μ values lead to larger support sets, which can make further calculations more difficult. The value for μ should be chosen by balancing these two objectives [24]. Our choice for the value of μ in the discretization of the six example data sets was guided by the range of μ values that Hammer et al. proposed in [56], namely between 3 and 10. We then tried to pick a value as high as possible such that further calculations are still able to run. The values we chose for μ in the greedy procedure are listed in Table 3.1.

The feature selection step is computationally expensive. The running time including all calculations for parameter vectors and matrices takes around 6 hours for the bigger problems such as the HOUS, PID and the CRED data set. In the last step of the workflow, a binary file including only the selected features is generated by `PickFeatures.py`. This data file can then be used for further calculations.

Input data

The programs for pattern generation and theory formation in `AnswerSetLAD` are implemented using ASP. Hence the data has to be in a suitable format.

A binary data file generated by the workflow described in the previous section or any other `csv`-file in the correct format can be translated into an ASP input file using the python script `MakeDataFile.py`. For this purpose the `csv`-file needs to include one row for each observation. The first column contains 0 or 1 depending on whether the observation is negative or positive. The following columns contain the binary values of the variables. An example of a data input translated for ASP is shown in Figure 3.3.

For each entry of the input a predicate `i(Sign,ID,Variable,Value)` is given, where `Sign` is the sign of the observation (1 for positive and 0 for negative), `ID` is the identifier of the observation and `Value` is the binary value of a `Variable`.

After having generated a suitable input data file we can proceed to the pattern generation step.

3.3.2 Pattern generation

Once the input data is transformed into a suitable file, the pattern generation process can be started. `AnswerSetLAD` provides programs to calculate several types of patterns. Besides patterns without any specified preference, called *general patterns* in the following, these pattern types are *prime patterns*, *spanned patterns*, *strong patterns*, *maximal patterns*, *strong prime patterns*, *strong spanned patterns* and *maximal prime patterns*.

```
1,0,0,1,0,1,1
1,1,1,1,1,0,1
1,0,1,1,0,0,1
1,0,1,0,1,1,0
0,0,0,0,1,1,0
0,1,1,0,0,0,1
0,1,0,1,0,1,1
```

```
i(1,1,1,0). i(1,1,2,0). i(1,1,3,1). i(1,1,4,0). i(1,1,5,1). i(1,1,6,1).
i(1,2,1,1). i(1,2,2,1). i(1,2,3,1). i(1,2,4,1). i(1,2,5,0). i(1,2,6,1).
i(1,3,1,0). i(1,3,2,1). i(1,3,3,1). i(1,3,4,0). i(1,3,5,0). i(1,3,6,1).
i(1,4,1,0). i(1,4,2,1). i(1,4,3,0). i(1,4,4,1). i(1,4,5,1). i(1,4,6,0).
i(0,5,1,0). i(0,5,2,0). i(0,5,3,0). i(0,5,4,1). i(0,5,5,1). i(0,5,6,0).
i(0,6,1,1). i(0,6,2,1). i(0,6,3,0). i(0,6,4,0). i(0,6,5,0). i(0,6,6,1).
i(0,7,1,1). i(0,7,2,0). i(0,7,3,1). i(0,7,4,0). i(0,7,5,1). i(0,7,6,1).
```

Fig. 3.3.: Exemplary data set in `csv`-format and corresponding input data for `AnswerSetLAD` generated by `MakeDataFile.py`

The ASP programs can be divided into two approaches depending on the basis of their encoding. A schematic overview of the hierarchy of the encodings for pattern generation is shown in Figure 3.4. While general patterns, prime patterns and maximal patterns belong to the approach of *literal-based pattern generation*, strong and spanned patterns belong to the approach of *coverage-based pattern generation*. This division is due to the definitions of the pattern preferences (see Definition 2.3 to 2.5). Prime patterns have an inclusion-wise minimal set of *literals*. This naturally leads to a program including rules to choose a suitable set of literals. For clarification, we remark here that in this case and in the following text we refer to literals with respect to the LAD terminology and not to the atoms and negated atoms of the ASP program if not explicitly stated differently.

In contrast, strong patterns and spanned patterns are defined with respect to their coverage. For this reason, we base the programs for strong and spanned pattern generation on the set of covered observations. We explain the two classes for pattern generation and the programs themselves in detail in the following subsections.

Literal-based pattern generation

We first describe the ASP encodings that belong to the class of literal-based pattern generation.

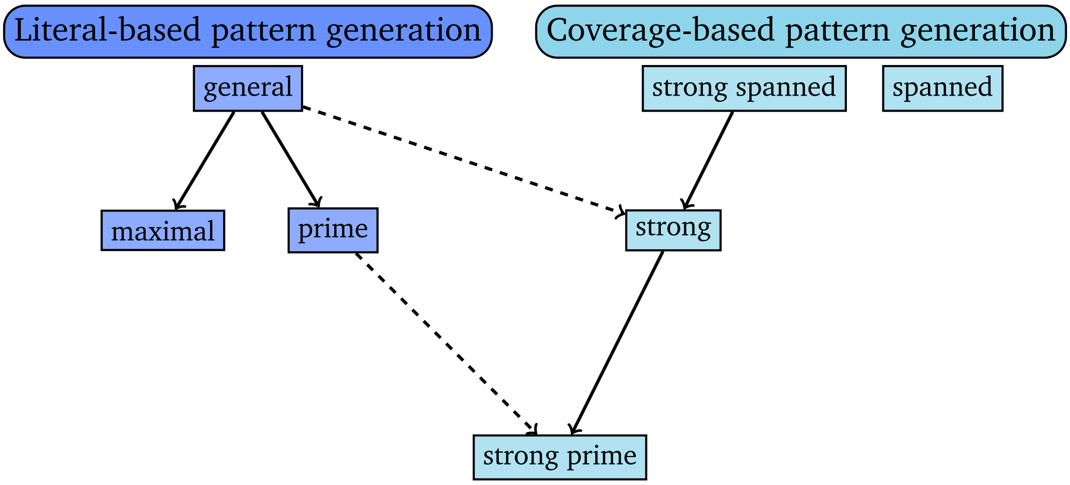

Literal-based pattern generation
Coverage-based pattern generation
general
strong spanned
spanned
maximal
prime
strong
strong prime

Fig. 3.4.: Schematic overview of the hierarchy of the encodings for pattern generation in `AnswerSetLAD`.

General patterns The encoding for general pattern generation is given in Figure 3.5. This is the basic program that calculates all patterns without any specified preference. Any pattern of any type obviously is included in the set of stable models of this program. Therefore, we can generate all different types of patterns by using this encoding and narrowing down the answer set to the desired definition of the pattern type. We describe this procedure in more detail in the later part of this section.

Before running the program, the user defines the four constants `degree`, `sign`, `homogeneity` and `prevalence`. A stable model provides a pattern of given degree and sign with homogeneity and prevalence greater than or equal to the chosen constants.

The program is organized in three parts, namely generate, define and test. A stable model is a pattern of given degree. Any stable model thus includes `degree` many atoms over the predicate `pat(S,B)`, where `S` is the variable name and `B` its Boolean value. The first rule given in line 2 is a choice rule generating solution candidates. We know from the definition of a pattern that it has to cover at least one observation of its sign. Therefore, we choose literals that cover the observations of the same sign. At this point, it is not given that the set of literals belongs to an actual pattern. It could be any set of single literals that cover different observations of the correct sign, but such that the whole set of literals is not covering any of the observations.

The define part is used to specify predicates that narrow the stable model down to the answer set we want to generate. To ensure that the pattern in the answer set covers the desired number of observations of the correct sign and not more than the allowed number of observations of the opposite sign, we introduce the predicate

```
1  % GENERATE
2  degree { pat(S,B) : i(sign,_,S,B) } degree.
3
4  % DEFINE
5  not_covered(W,X) :- i(W,X,_,_), pat(S,B), not i(W,X,S,B).
6  covered(W,X) :- not not_covered(W,X), i(W,X,_,_).
7
8  % TEST
9  :- pat(S,B), pat(S,Q), Q<B.
10 :- #sum{ homogeneity-100,X : covered(W,X), W=sign;
11         homogeneity,X : covered(W,X), W!=sign } > 0.
12 :- nbrcorrectobs(C),
13         #sum{ 100,X : covered(W,X), W=sign } < prevalence*C.
```

Fig. 3.5.: Basic ASP encoding for general pattern generation

covered(W,X) in line 6, which is true if observation X having sign W is covered by the chosen set of literals. For defining this predicate, we use the auxiliary predicate not_covered(W,X) in line 5, which is true for an observation W if one of the literals pat(S,B) does not cover W. Then covered(W,X) is true if not_covered(W,X) is not true.

In the test part we can make use of the defined predicate and test whether the choice of literals fulfills the definition of a pattern with given homogeneity and prevalence. Line 9 is a general test forbidding that a pattern contains the same variable with different assignments. In line 10 and 11, we test whether the set of literals fulfills the homogeneity condition (Definition 2.12). In addition, line 12 and 13 make sure that the prevalence condition (Definition 2.13) is met by the generated pattern. The unary predicate nbrcorrectobs is calculated in advance and counts the number of observations having the same sign as the pattern wanted.

One can argue whether it is useful to define the used parameters as input constants for the program. This of course depends on the output we want to generate. The constant sign seems to be a reasonable input as the generation of positive and negative patterns is symmetric and the two resulting sets of patterns are used in opposite ways, namely for the characterization of the set of positive and negative observations, respectively. The parameters homogeneity and prevalence can be used to reduce the size of the pool of answer sets. The set of patterns for a data set is huge in general. Especially when talking about patterns without specified preference the pool of answer sets becomes unmanageable large. Therefore, we find it useful to include these parameters in advance. We added the constant degree here for a similar reason. As the pool of answers gets large it is helpful to specify the answer

```
1  % DEFINE
2  covered_after_deletion(W,X,S) :- i(W,X,S,_),
3          #count{ T : pat(T,C), i(W,X,T,C), T != S } = degree-1.
4
5  % TEST
6  :- not covered(sign,_).
7  :- pat(S,B), #sum{ homogeneity-100,X : covered_after_deletion(W,X,S),
8          W = sign; homogeneity,X : covered_after_deletion(W,X,S),
9          W != sign } <= 0.
```

Fig. 3.6.: Additional lines to the encoding in Figure 3.5 for the generation of prime patterns.

one is looking for. If one wants to calculate all patterns of any degree this constant can just be left out from the input. Then the first row simply gets replaced by the following choice rule:

```
1{pat(S,B):i(sign,_,S,B)}.
```

The degree of the pattern can then be determined in a later step if needed. The constants `homogeneity` and `prevalence` can be deleted including the last two integrity constraints if they are not of interest. They then have to be replaced by the rules coming from the original definition of patterns saying that at least one observation of correct sign has to be covered and no observation of opposite sign is allowed to be covered. We have used this setup for some of our applications and later programs as we will see in the further paragraphs.

Prime patterns By expanding the encoding presented above by only one more test, we can constrain the set of patterns to those which are prime. The additional lines presented in Figure 3.6 are based on the definition of prime patterns. A pattern is prime if and only if the deletion of any of its literals results in a term that is not a pattern. We determine the coverage of an observation `X` of sign `W` for each term obtained by the deletion of a single literal `S` (lines 2-3) within the predicate `covered_after_deletion(W,X,S)`. To ensure that the pattern covers at least one observation, we use line 6. Then we test whether the term satisfies the homogeneity condition as described in the program above (lines 7-9). Note that it is not necessary to test the prevalence condition as a term obtained by leaving out one of the literals cannot cover less observations than the original pattern. A pattern that still has a homogeneity above the given constant after the deletion of a literal is not a prime pattern.

Maximal patterns The calculation of maximal patterns is realized by adding only one additional line to the basic encoding given in Figure 3.5. Maximal patterns are patterns that have a maximal number of covered observation (Definition 2.11). For this attempt we simply use the following maximize-statement at the end of the encoding:

```
#maximize{1,X : covered(W,X) }.
```

Coverage-based pattern generation

While the implementations of the three pattern types shown above are based on stable models of LAD literals the following implementations yield stable models of covered observations. Whereas prime patterns are defined with respect to their literal set, the definition of spanned patterns and strong patterns is built on the set of covered observations. A construction starting from the definitions of the pattern types thus naturally leads to a different focus in the modeling.

Other than the literal-based encodings the coverage-based encodings, which we show in this subsection, generate patterns of any degree, any homogeneity and any prevalence (except for strong and strong prime patterns). This means that `degree`, `homogeneity` and `prevalence` do not appear as constants given to the program by the user. Regarding our applications it turned out that we were interested in the whole set of patterns of a specific type having any degree and any homogeneity and prevalence. Therefore, for the following encodings the only parameter that has to be given in advance remains the `sign` of the patterns. We discussed before that this is a different approach, which could make sense depending on the desired result. Of course, this can easily be modified by constraining the choice rules to a size of `degree` and adding two more integrity constraints for `homogeneity` and `prevalence` as seen before.

Spanned patterns In Figure 3.7 the encoding for the generation of spanned patterns is shown. A pattern is called spanned if it is Pareto-optimal with respect to the selectivity and the evidential preference.

The answer set consists of covered observations of sign. Implicitly the program thereby calculates the pattern that covers this set of observations. The first rule given in line 2 is a choice rule picking solution candidates from the total set of observations. The number 1 on the left hand side of the brackets indicates that at least one observation has to be covered.

```
1  % GENERATE
2  1 { cov(sign,X) : i(sign,X,_,_) }.
3  nbrcovered(N) :- N=#sum{ 1,X : cov(sign,X) }.
4
5  % DEFINE
6  lit_candidate(S,B) :- cov(sign,X), i(sign,X,S,B).
7  not_lit(S,B) :- lit_candidate(S,B), cov(sign,Y), not i(sign,Y,S,B).
8  lit(S,B) :- not not_lit(S,B), lit_candidate(S,B).
9  countlit(E) :- E=#sum{ 1,(S,B) : lit(S,B) }.
10
11 in_opposite_obs(Y,(S,B)) :- lit(S,B), i(Q,Y,S,B), Q!=sign.
12 countinop(Y,D) :- i(Q,Y,_,_), Q!=sign,
13         D=#sum{ 1,(S,B) : in_opposite_obs(Y,(S,B)) }.
14
15 obsnotincover(sign,Y) :- i(sign,Y,_,_), not cov(sign,Y).
16 not_addobs(sign,Y) :- obsnotincover(sign,Y), lit(S,B),
17         not i(sign,Y,S,B).
18 addobs(sign,Y) :- obsnotincover(sign,Y), not not_addobs(sign,Y).
19
20 % TEST
21 :- D=E, countlit(E), countinop(_,D).
22 :- addobs(sign,_).
```

Fig. 3.7.: Encoding for the generation of spanned patterns

The define part is used to narrow down the chosen set to an accurate answer, namely a set of covered observations with the according spanned pattern. Sets of pattern literals that are able to cover the chosen set of observations are generated using the following definitions. In line 6 the predicate `lit_candidate(S,B)` is defined by each literal that covers a covered observation, where `S` is the variable name and `B` its Boolean value. But not all of these candidates are literals of a pattern. Only those literals that cover every covered observation can be included in a pattern. For this purpose, in lines 6-8 the literals `lit(S,B)` are calculated as the subset of the literal candidates `lit_candidate(S,B)` that are not *not literals* `not_lit(S,B)`. A *not literal* here is a literal that appears in the set of literal candidates but there exists an observation included in the chosen set of covered observations that is not covered by this literal (line 7).

At this point of the program we found a combination of literals that covers all the chosen covered observations, but we do not know yet whether this literal combination covers an observation of the opposite sign. To make sure that it does not and that the literal combination as a result is a pattern, the definitions in line 11-12 are needed. In the predicate `in_opposite_obs(Y,(S,B))` all literals `lit(S,B)` are collected together with the observation Y of the opposite sign that the literal is covering. The number D of literals that cover an observation Y of opposite sign are counted in the predicate countinop(Y,D) (line 12-13). This predicate can then be used in the test part to ensure that none of the observations of opposite sign is covered by all literals of the generated term (line 21). In that case the term is a pattern. More precisely it is a pattern which is selectivity-wise preferred.

For the evidential preference of the pattern we use the definitions in lines 15-18. We have to make sure that the set of covered observations chosen is as large as possible for the set of literals. Therefore, we collect all observations of the correct sign which are not in the chosen set of covered observations with the predicate obsnotcover(sign,Y) (line 15). We cannot add an observation Y to the set of chosen covered observations if a literal of the pattern does not cover this observation (lines 16-17). Symmetrically we *can* add the observation if we cannot *not* add it to the chosen set (line 18). If this is possible, then the pattern formed by the set of literals is not spanned, because it is not evidentially preferred. Therefore, we exclude these results from the set of answers (line 22). A stable model of the program is a spanned pattern together with the set of observations that it covers.

Strong Spanned patterns Strong spanned patterns are those patterns that are Pareto-optimal with respect to the lexicographic refinement of the evidential preference by the selectivity preference. They are the strong patterns having an inclusion-wise maximal set of literals. For this reason, the structure of the encoding resembles the one calculating spanned patterns. The whole program is shown in Figure 3.8. We here explain the parts in which this program differs from the one before.

The first 13 lines are exactly the same as in the program explained before. A set of covered observations is chosen and the set of literals of the pattern is defined by the set of literals which cover all of the chosen observations.

Within the following lines we define predicates to check whether it is possible to add an observation of the correct sign to the set of chosen observations. This is the evidential preference.

For this purpose, we first define the predicate lit_in_new(Y,(S,B)), which is used to show which of the literals (S,B) that where already chosen also covers an observation Y of correct sign that is not in the set of chosen covered observations (line 15). We use the same predicate structure as before for the original literals lit(S,B) to test whether this subset of literals also covers an observation of opposite sign. In detail this means that we define for every pair of observation Z of the opposite

```
 1  % GENERATE
 2  1 { cov(sign,X) : i(sign,X,_,_) }.
 3  nbrcovered(N) :- N=#sum{ 1,X : cov(sign,X) }.
 4
 5  % DEFINE
 6  lit_candidate(S,B) :- cov(sign,X), i(sign,X,S,B).
 7  not_lit(S,B) :- lit_candidate(S,B), cov(sign,Y), not i(sign,Y,S,B).
 8  lit(S,B) :- not not_lit(S,B), lit_candidate(S,B).
 9  countlit(E) :- E=#sum{ 1,(S,B) : lit(S,B) }.
10
11  in_opposite_obs(Y,(S,B)) :- lit(S,B), i(Q,Y,S,B), Q!=sign.
12  countinop(Y,D) :- i(Q,Y,_,_), Q!=sign,
13          D=#sum{ 1,(S,B) : in_opposite_obs(Y,(S,B)) }.
14
15  lit_in_new(Y,(S,B)) :- lit(S,B), i(sign,Y,S,B), not cov(sign,Y).
16  nbrlitinnew(Y,M) :- lit_in_new(Y,_),
17          M=#sum{ 1,(S,B) : lit_in_new(Y,(S,B)) }.
18
19  litinnew_in_op(Z,(S,B),Y) :- lit_in_new(Y,(S,B)), i(Q,Z,S,B), Q!=sign.
20  litinnew_countinop(Z,D,Y) :- i(Q,Z,_,_), Q!=sign,
21          litinnew_in_op(Z,_,Y),
22          D=#sum{ 1,(S,B) : litinnew_in_op(Z,(S,B),Y) }.
23  notpat(Y) :- nbrlitinnew(Y,M), litinnew_countinop(Z,D,Y), M=D.
24
25  % TEST
26  :- D=E, countlit(E), countinop(_,D).
27  :- not notpat(Y), lit_in_new(Y,_).
```

Fig. 3.8.: The encoding for the generation of strong spanned patterns

sign and observation Y of correct sign which is not in the original chosen set of observations the predicate litinnew_in_op(Z,(S,B),Y) that exists if the literal (S,B) covers the observation Z (line 19). In the following lines 20-22 we first count the D literals which cover the observation Z of opposite sign and then compare this number to the full number of literals which cover all chosen observations plus the observation Y in line 23. If those two numbers are the same, meaning that the whole term covers an observation of opposite sign, then the term is not a pattern. This implies that we do not find a pattern which covers the chosen set of observations plus the observation Y. In that case, the predicate notpat(Y) is true. Within the test part we then say that any chosen set of observations and the associated set of literals is not an answer set, or strong spanned pattern, if it is possible to add an observation Y of correct sign to the set of covered observations and find a pattern which covers them all (line 27). The resulting answer sets are all strong spanned patterns.

```
 1  % GENERATE
 2  1 { pat(S,B) : lit(S,B) }.
 3
 4  % DEFINE
 5  not_covered(W,X) :- i(W,X,_,_), pat(S,B), not i(W,X,S,B).
 6  covered(W,X) :- not not_covered(W,X), i(W,X,_,_).
 7
 8  % TEST
 9  :- pat(S,B), pat(S,Q), Q<B.
10  :- #sum{ homogeneity-100,X : covered(W,X), W=sign;
11          homogeneity,X : covered(W,X), W!=sign } > 0.
12  :- nbrcorrectobs(C), #sum{ 100,X : covered(W,X), W=sign }
13          < prevalence*C.
```

Fig. 3.9.: Additional lines to the encoding in Figure 3.8 for the generation of strong patterns

Strong patterns The encoding for strong patterns builds on how strong patterns are defined in relation to strong spanned patterns. The additional lines of code we need for expanding the code in Figure 3.8 in order to get stable models that represent the strong patterns is shown in Figure 3.9. We noted before that strong spanned patterns are a subset of strong patterns. While the property *spanned* ensures that a stable model of the encoding for the strong spanned pattern calculation in Figure 3.8 consists of an inclusion-wise maximal number of literals, the full set of strong patterns are all possible combinations of literals that form a pattern for an inclusion-wise maximal set of covered observations. Therefore, we can generate the set of strong patterns out of the set of strong spanned patterns by combining the computed set of literals in every possible way such that the resulting combination remains a pattern, i.e., does not cover an observation of the opposite sign (or does not contradict the chosen bounds on homogeneity and prevalence). The additional lines given in Figure 3.9 thus are the same as the encoding for general patterns in Figure 3.5 except for line 2 where the predicate `pat(S,B)` is chosen out of the already calculated set of literals instead of all possible literals for a data set.

Strong prime patterns The encoding for strong prime pattern generation is based on the calculation of strong patterns, as the strong prime patterns are a subset of the strong patterns for a data set. We show the additional lines to the encoding in Figure 3.9 in Figure 3.10. They very closely resemble the already introduced code for the generation of prime patterns (see Figure 3.6). The only difference is that in line 2 the degree `D` has to be counted and stored in the predicate `countdegree(D)` because the degree is not given to the program as a constant.

```
% DEFINE
countdegree(D) :- D=#sum{ 1,(S,B) : pat(S,B) }.

covered_after_deletion(W,X,(S,B)) :- i(W,X,_,_), pat(S,B),
        not i(W,X,S,B), #sum{ 1,(T,C) : pat(T,C), i(W,X,T,C),
        (T,C)!=(S,B) }= D-1, countdegree(D).

homgood(S,B) :- pat(S,B), #sum{ homogeneity-100,X:
        covered_after_deletion(W,X,(S,B)) ,W=sign;
        homogeneity,X : covered_after_deletion(W,X,(S,B)), W!=sign }
        <= 0.

% TEST
:- homgood(S,B).
```

Fig. 3.10.: Additional lines to the encoding in Fig. 3.9 for the generation of strong prime patterns

Using asprin for the calculation of Pareto-optimal patterns

Within the last paragraphs of this chapter we described in detail our implementation using ASP for the generation of the various different Pareto-optimal pattern types defined by Hammer et al. [57]. For each pattern type we presented a program coding the desired preference via hard constraints. These programs can then be grounded and solved by the ASP solver and grounder. We used clingo [49] for all our experiments, but various other systems for ASP grounding and solving are available too.

Within Potassco [33], the Potsdam Answer Set Solving Collection, several systems besides clingo have been developed for specific purposes. One of them is asprin [29]. While clingo includes functionalities for single objective and lexicographic optimization with priorities or weights, the framework asprin allows the computation of optimal answer sets with preferences. Some commonly used preference types like subset minimality and Pareto-optimality are defined in the asprin library. Besides that, new preferences can be added according to the individual aims of the user. In the following text, we give an insight into the use of asprin with respect to the functionalities needed for our intent. For a detailed description we refer the reader to [29] or [48].

Making use of asprin, the different pattern types can easily be implemented based on the encoding for general pattern generation in Figure 3.5, varying only the preference specifications at the end of the logic program. The three preferences used for LAD patterns (see Section 2.3.1) can be implemented using asprin and based on the predicates in the encoding for general patterns in Figure 3.5 in the following way:

- The simplicity preference:

  ```
  #preference(simplicity,subset){pat(S,B)}.
  ```

- The selectivity preference:

  ```
  #preference(selectivity,superset){pat(S,B)}.
  ```

- The evidential preference:

  ```
  #preference(evidential,superset){covered(sign,X)}.
  ```

The preference specifications are indicated by #preference. The name of each preference is given first, here simplicity, selectivity and evidential, followed by the type of the preference. The types we use are subset and superset. Lastly the predicate for the preference is given. In the simplicity preference specification, for example, we define the preference called simplicity of type subset over the atoms of the predicate pat/2. For the specification of the optimization objective an optimization statement has to be added to the end of the program, which includes the name of the preference to be optimized.

For the LAD prime pattern generation we, therefore, expand the encoding for general pattern generation in Figure 3.5 using asprin by the following lines:

```
#preference(simplicity,subset){pat(S,B)}.
#optimize(simplicity).
```

The program is modified to calculate strong patterns by adding:

```
#preference(evidential,superset){covered(sign,X)}.
#optimize(evidential).
```

In the asprin framework preferences can also be combined using different principles. We here make use of two of them, which are pareto and lexico, referring to Pareto-optimality and optimality with respect to lexicographic refinement, respectively.

For the generation of spanned patterns we write the following lines:

```
#preference(selectivity,superset){pat(S,B)}.
#preference(evidential,superset){covered(sign,X)}.

#preference(spanned,pareto){**selectivity;**evidential}.
#optimize(spanned).
```

Strong prime and strong spanned patterns are defined using lexicographic refinement. This leads to the following lines for the implementation of strong prime patterns:

```
#preference(simplicity,subset){pat(S,B)}.
#preference(evidential,superset){covered(sign,X)}.

#preference(strongprime,lexico){1::**simplicity;2::**evidential}.
#optimize(strongprime).
```

The order of the preferences has to be defined using weights for the preference types in the `lexico` preference. Here the evidential preference has higher weight and, therefore, higher priority than the simplicity preference. This notation of `asprin` is orientated towards the syntax and semantics of `clingo`. In a similar way the strong spanned patterns can be encoded:

```
#preference(selectivity,superset){pat(S,B)}.
#preference(evidential,superset){covered(sign,X)}.

#preference(strongspanned,lexico){1::**selectivity;2::**evidential}.
#optimize(strongspanned).
```

We showed in the former paragraphs how `asprin` can be used for a succinct implementation of the pattern types with preferences in LAD. With respect to clarity and comprehensibility of the encodings the use of `asprin` is definitely favorable compared to the encodings shown in the previous subsection. However, we also have to take the performance regarding running time of the implementations with and without `asprin` into account. We ran tests on the six data sets from the UC Irvine Machine Learning Repository [38] introduced in the former section (see Table 3.1). We used the data sets after binarization and feature selection. In this study, we compared the running time of the calculations for *all* patterns of a pattern type by the implementations explained before, using hard constraints and *no* `asprin`, with the running time for the same calculations by the implementation using `asprin` as just described. The results are shown in Table 3.2. We present the running time for the given task. In parentheses the number of patterns calculated is listed. The faster of the two approaches is marked in green for each task.

The first remark on the running times of the `asprin` versus the *no* `asprin` implementations is that we do not see a clear answer to the question which of the two solves the given problems faster. Only for the generation of spanned patterns our implementation using hard constraints has a shorter running time than the `asprin` approach in all experiments. For each of the other pattern types it seems to depend on the underlying data set whether the one or the other approach yields a faster result.

For both approaches the running time mostly goes up when a lot of patterns have to be calculated, which makes sense because of the time needed for enumeration. Taking a closer look at the spanned pattern calculation we think that here the *no* `asprin` implementation has an advantage because a lot of patterns are calculated by `asprin` that are not optimal. For the CRED data set, for example, `asprin` generated 2809 models in total out of which only 672 are optimal. The implementation not using `asprin` does not use optimization at all and might, therefore, be faster when only enumerating answer sets.

For all pattern types including the evidential preference, namely strong, strong prime and strong spanned patterns, the experiment suggests that for smaller data sets (at the end of the table) our original implementations lead to less running time while for the larger data sets (at the beginning of the table) the `asprin`-based approach takes over. This behavior could be due to the more restricted search space via the implementations using hard constraints. A more restricted search space leads to faster results for small instances. However, the `asprin`-based approach uses significantly less rules, which leads to less overhead and rules after grounding. This makes the implementation using `asprin` preferable for larger instances.

In this subsection, we explained in detail all ASP-encodings for pattern generation within our software package `AnswerSetLAD`. The calculated patterns or sets of patterns carry a lot of useful information about the characteristics of a data set and its positive and negative class of observations. They are, therefore, of interest on their own. However, if one wants to make predictions for the class of a new observation a theory has to be built. In the following subsection we address this step of LAD and its implementation in `AnswerSetLAD`.

	prime		strong		spanned	
	asprin	no asprin	asprin	no asprin	asprin	no asprin
BCW	13.9s (95)	10.7s (95)	1.7s (8407)	9.3s (8407)	4d14h06m0.3s (37758)	6.4s (37758)
HD	2.9s (100)	1.5s (100)	0.7s (145)	1.7s (145)	38.2s (493)	0.4s (493)
BLD	0.7s (9)	2.6s (9)	2.3s (28800)	1m10.2s (28800)	1.0s (6)	0.1s (6)
CRED	2m44.6s (324)	35.7s (324)	2.8s (37656)	29.6s (37656)	5m58.0s (672)	0.5s (672)
PID	41.0s (131)	5.2s (131)	44.8s (955400)	38.9s (955400)	14.1s (89)	0.1s (89)
HOUS	6.7s (70)	0.9s (70)	0.5s (1120)	0.1s (1120)	2.7s (27)	0.0s (27)

	strong prime		strong spanned	
	asprin	no asprin	asprin	no asprin
BCW	7.4s (27)	13.3s (27)	2.2s (26)	1m10.6s (26)
HD	2.8s (34)	3.9s (34)	1.4s (29)	2.6s (29)
BLD	0.5s (4)	24.3s (4)	0.8s (4)	19.9s (4)
CRED	20.7s (68)	38.2s (68)	3.6s (32)	32.6s (32)
PID	0.9s (7)	1.9s (7)	1.0s (6)	0.4s (6)
HOUS	0.5s (6)	0.3s (6)	0.3s (2)	0.0s (2)

Tab. 3.2.: Results of the running times (CPU) of the implementations for the generation of *all* patterns of the different pattern types using `asprin` and *no* `asprin` on the six data sets taken from [38]. (See Table 3.1 for more details on the data sets.) The number of patterns calculated is shown in parentheses.

3.3.3 Theory formation and prediction

The last step in the LAD workflow and the AnswerSetLAD package is the theory formation and the associated prediction for new observations. The theory formation step is implemented in AnswerSetLAD using both python and ASP code.

An iterative algorithm for the calculation of a pattern cover

In the introductory chapter on LAD, Chapter 2, we showed the algorithm proposed by Hammer et al. [35] and revisited by Ryoo et al. [106] for the successive selection of patterns based on a data set to form a theory (see Chapter 2 Algorithm 2). The algorithm is based on the principle to consecutively choose a pattern that covers a (maximal) set of observations, which are then deleted from the input set of observations. This is done repeatedly until enough patterns are chosen such that each observation is covered. We implemented the procedure within AnswerSetLAD in python under the name PattGen.py following the naming of [106]. The python implementation needs two ASP files as input, which are the binary data file and the pattern type that should be used for the cover in form of the ASP program. Up to now the implementation is designed to use maximal patterns only following the proposed algorithm in [106] but it can easily be extended to take different pattern types as input. In that case one has to be aware that not all pattern types can be used here as it is not given that each observation of a data set is covered by a pattern of each type, which is crucial for the presented algorithm to finish.

An exemplary program call for the binary BCW data set after feature selection looks as follows:

```
python PattGen.py BCW_selectedfeatures.asp AnswerSetLAD_maximal.asp
```

The program PattGen.py subsequently produces the following output:

```
--- PattGen generates a full pattern cover for you. ---
data file:  BCW_selectedfeatures.asp
pattern type used for cover: AnswerSetLAD_maximal.asp

--- Positive patterns ---
pattern: ['pat(14,1)']
size of coverage: 95
```

```
pattern: ['pat(13,1)', 'pat(16,1)']
size of coverage: 26
pattern: ['pat(17,1)', 'pat(15,1)']
size of coverage: 13
pattern: ['pat(11,0)', 'pat(20,1)']
size of coverage: 7
pattern: ['pat(18,1)', 'pat(19,1)']
size of coverage: 5
pattern: ['pat(10,1)', 'pat(18,1)']
size of coverage: 1
pattern: ['pat(20,0)', 'pat(7,0)', 'pat(6,0)',
 'pat(16,0)', 'pat(17,0)', 'pat(18,0)']
size of coverage: 1
pattern: ['pat(1,1)']
size of coverage: 1
pattern: ['pat(9,1)', 'pat(6,0)', 'pat(17,0)']
size of coverage: 1
pattern: ['pat(12,1)']
size of coverage: 1
pattern: ['pat(9,1)', 'pat(16,1)']
size of coverage: 1
--- Negative patterns ---
pattern: ['pat(1,0)']
size of coverage: 9
------------------------------------------------------
--- Done in 0.270208120346 seconds.
------------------------------------------------------
```

As mentioned before the algorithm and its implementation can be used to find a full pattern cover for a data set. However, there might be more reasonable ways to choose patterns than by their coverage. This part of our software is still expandable and leaves room for prospective work.

Another highly interesting topic, which is solvable in diverse ways, is if and how weights should be chosen when building a discriminant based on a pattern cover as described in Chapter 2 Section 2.4.2. Several ideas are given in [24], which all depend on the application and the underlying data.

We will address the topic of theory formation more closely in terms of theoretical ideas in Chapter 4. We refer the interested reader to this part of the thesis for an

ASP approach to build a prime theory out of the set of all prime patterns for a data set. For this specific class of theories, AnswerSetLAD contains an ASP program called AnswerSetLAD_predict.asp that predicts the class of a new observation based on whether it is covered by more positive or more negative patterns.

3.4 Performance testing

In the previous section, we presented the implementation of the LAD functionalities in our software package AnswerSetLAD. Besides the declarative style of the programs, which allows easy handling and maintenance, one of our main goals was runtime efficiency on large data sets. In this section, we test the performance of AnswerSetLAD exemplarily with respect to the generation of maximal patterns by comparing our approach to Mixed-Integer Linear Programming (MILP). MILP is widely used for the calculations of patterns in LAD.

3.4.1 Comparison to a Mixed-Integer Linear Programming approach

Apart from the clear language of ASP our software AnswerSetLAD provides a promising alternative to the commonly used Mixed-Integer Linear Programs (MILP) for pattern generation regarding running time. To substantiate this statement, we ran experiments on the six data sets BCW, HD, BLD, HOUS, CRED and PID, which we discretized and discussed before in Section 3.3.1. These data sets are widely used in the LAD literature (see [24, 56, 93, 106] for example).

The MILP

For this study we used the MILP proposed by Ryoo et al. [93] for the generation of maximal patterns. The MILP is shown in Chapter 2 Figure 2.3.

In order to bring the program into a format that is accessible for a MILP solver we used the algebraic modeling language ZIMPL [108] by Thorsten Koch [65]. The MILP written in ZIMPL is shown in Figure 3.11.

Depending on the underlying data sets the input for the MILP and with it the parameters have to be adjusted. We see an example for the input of the introductory data set for LAD on the patient having stomach ache (see Chapter 2 Table 2.1) in Figure 3.12. The data set consists of 6 attributes and 7 observations. The class of each

```
#GENERAL SETS
#index sets (rows*columns)
#rows
set I := { 1 .. m };
#columns
set J := { 1 .. n };
#index set for negated parameters
set K := { n+1 .. 2*n };

#MILP
var Y[Positive] binary;
var X[J] binary;
var Xneg[K] binary;
var degree integer >= 1 <= n ;

minimize notcovered: sum <k,l> in Positive: Y[k,l];

subto onlyone: forall <j> in J do
X[j] + Xneg[j+n] <= 1 ;

subto patterndegree: sum <j> in J : X[j] + sum <k> in K :
Xneg[k] == degree;

subto opposallnotcovered: forall <i,l> in Negative do
sum <j> in J : (a[i,j] * X[j]) + sum <k> in K :
(aneg[i,k] * Xneg[k]) <= degree - 1;

subto atleastonecovered: forall <i,l> in Positive do
sum <j> in J : a[i,j] * X[j] + n * Y[i,l] + sum <k> in K :
aneg[i,k] * Xneg[k] + n * Y[i,l] >= degree;
```

Fig. 3.11.: MILP approach for maximal pattern calculation by Ryoo et al. [93] notated using ZIMPL [108].

observation is documented in the set Classes. The matrix param a[I*J] consists of the attribute values belonging to the seven given observations. The MILP also needs the negated attribute values, which are given in the matrix param aneg[I*K]. For our purposes, we wrote a python script to parse a csv-data file to a MILP input file as shown in Figure 3.12.

Setup

To compare the run time efficiency of AnswerSetLAD and the MILP approach proposed in [93] for the generation of maximal patterns, we used the six data sets taken from the UC Irvine Machine Learning Repository [38], which we introduced in Section 3.2. For reasons of comparability we chose to do tests both on the full

```
#PARAMETERS
#number of attributes
param n := 6;
#number of observations
param m := 7;

#DATA
#classes
set Classes := {<1,1>,<2,0>,<3,1>,<4,1>,<5,0>,<6,1>,<7,0>};

#positive and negative observations
set Positive := { <k,l> in Classes with l == 1 };
set Negative := { <k,l> in Classes with l == 0 };

#attribute values
param a[I*J] :=
|1,2,3,4,5,6|
|1|0,0,1,0,1,1|
|2|0,0,0,1,1,0|
|3|1,1,1,1,0,1|
|4|0,1,1,0,0,1|
|5|1,1,0,0,0,1|
|6|0,1,0,1,1,0|
|7|1,0,1,0,1,1|;

#negated attribute values
param aneg[I*K] :=
|7,8,9,10,11,12|
|1|1,1,0,1,0,0|
|2|1,1,1,0,0,1|
|3|0,0,0,0,1,0|
|4|1,0,0,1,1,0|
|5|0,0,1,1,1,0|
|6|1,0,1,0,0,1|
|7|0,1,0,1,0,0|;
```

Fig. 3.12.: Example of an input based on the data set shown in Chapter 2 Table 2.1 for the MILP in Figure 3.11 notated using ZIMPL [108]

binary data sets consisting of all the level variables and the alternative binary data sets including only the selected features after the greedy selection process (see Table 3.1).

With respect to the standard workflow of LAD any non-binary data set is preprocessed including binarization and feature selection. The second step is crucial to make the data set accessible for further calculations as discussed before. This reason, and the fact that the authors in [93] and [106] did their calculations on data sets after feature selection, justifies our choice of the binary data sets including only the selected features. On the other hand the feature selection process includes several

	Binarized data sets after feature selections				
		MILP (solved by `SCIP`)		**ASP** (solved by `clingo`)	
	size	first	all	first	all
BCW	161 × 20	5.81s	7.92s	0.04s	0.04s
HD	73 × 11	0.74s	1.28s	0.01s	0.01s
BLD	341 × 15	0.06s	7.05s	0.05s	0.20s
CRED	114 × 19	0.84s	1.29s	0.03s	0.03s
PID	28 × 24	0.04s	0.14s	0.01s	0.01s
HOUS	14 × 20	0.02s	0.35s	0.00s	0.01s

Tab. 3.3.: Results for the running times (CPU) of the MILP and ASP approach for the calculation of positive maximal patterns on the six binarized data sets from the UC Irvine Machine Learning Repository [38] after feature selection.

parameters that can be adjusted according to the data set and the aims of the user. This leads to different features, which are selected, and, therefore, to different binary data sets. The setting for the parameters in the greedy procedure in [93] and [106] is not given in the respective articles. Hence, it was not possible for us to reconstruct the exact data sets. We tried to adjust our parameter settings in such a way that the number of resulting features is approximately the same as in the articles cited above.

Additionally we did all calculations on the binary data sets including all level variables before feature selection to allow results on data sets that are reproducible. The size of these data sets is comparatively large but does not depend on any parameters. Moreover, the larger data sets lead to longer running times for both the MILP and the ASP approach, as we discuss more closely in the following paragraphs, and show more nicely the gap between the two approaches.

Results

The test results are shown in Table 3.3 and Table 3.4. All calculations were made within a Linux AMD64 with 3.20GHz, 4 cores and 15.6GB of memory. The ASP programs were solved using the solver `clingo` [49]. The MILP were solved using `SCIP` [52] with the LP solver `SoPlex` [53, 41].

The times shown are the CPU times of the processes. For each of the six data sets we took the time to generate the first positive maximal pattern and the time to generate all positive maximal patterns both by the MILP and the ASP program. This was done for the smaller data sets with selected features (Table 3.3) and the larger sets including all level variables (Table 3.4), respectively. Note that the first positive

	Binarized original data sets (all level variables included)				
		MILP (solved by SCIP)		**ASP** (solved by clingo)	
	size	first	all	first	all
BCW	449 × 72	7m08s	34m52s	0.5s	0.5s
HD	297 × 305	7h39m	(≥ 72 hours)	6.0s	1h55m
BLD	341 × 269	5h32m	(≥ 72 hours)	3.6s	(≥ 72 hours)
CRED	653 × 773	(≥ 72 hours)	(≥ 72 hours)	17m28s	(≥ 72 hours)
PID	768 × 857	(≥ 72 hours)	(≥ 72 hours)	14h02m	(≥ 72 hours)
HOUS	506 × 1217	14h42m	(≥ 72 hours)	54.0s	(≥ 72 hours)

Tab. 3.4.: Results for the running times (CPU) of the MILP and ASP approach for the calculation of positive maximal patterns on the six binarized data sets from the UC Irvine Machine Learning Repository [38] before feature selection.

maximal pattern generated might differ between MILP and ASP. The results for all positive maximal patterns are the same in both cases.

While the ASP solver `clingo` provides the option to enumerate all optimal solutions using `-opt-mode=optN`, there is no direct approach for the enumeration of all optimal solutions implemented in `SCIP`. For the calculation of all positive maximal patterns with `SCIP` we, therefore, followed the instructions given in the `SCIP` documentation [101]. First the problem is solved to optimality. Second the objective function is added as a constraint to the MILP such that the objective function value has to be equal to the previously calculated optimal value. Next the predefined counting option in `SCIP` can be used to collect all feasible solutions for the adjusted MILP. For that reason, the time to find all positive maximal patterns with MILP, which is shown in Table 3.3 and Table 3.4, is the sum of the time to find an optimal solution and the time to collect all feasible solutions for the adjusted MILP.

Table 3.3, which includes the results for the small data sets with selected features, is in line with the results shown by Ryoo et al. in [106] regarding the timescale of seconds. All resulting times of the ASP code lie within milliseconds, getting bigger with data sets having more observations. Note here that the number of attributes is in the same range for all data sets in the feature selected case. The running times of the MILP lie above the times of the ASP program in all cases. However, this set of small benchmarks may not be representative for the overall performance.

Therefore, we evaluated the running times for the larger data sets shown in Table 3.4. These results clearly show the advantage of ASP. The number of observations in this table lies between 297 and 768 and the number of attributes goes up to 1217. The ASP program was able to solve all six tasks of finding one positive maximal pattern. It took less than a second for the full BCW data set and a few seconds for the larger sets regarding the number of attributes, which are HD and BLD. In comparison, the

MILP needed more than seven minutes for the easiest problem BCW and around five to eight hours for the data sets HD and BLD.

Interestingly the HOUS data set seems to be the easiest to solve out of the three larger sets CRED, PID and HOUS, although it includes more attributes. This might be due to the nature of the data itself but can also indicate that the difficulty depends on the number of observations more than on the number of attributes. The HOUS data set is in fact the only data set out of the bigger sets that the MILP was able to solve before the time-out. It took around 15 hours. The ASP program got the result for the same task in less than one minute.

For our calculations we used a time-out of 72 hours. The MILP did not finish the calculation of the first pattern on the CRED and the PID data set within this time frame. The ASP program was able to find an answer for the CRED data set in around 17 minutes and for the PID data set in around 14 hours.

By looking at the columns including the times to calculate all maximal patterns we see that both the MILP and ASP reach their limit. The MILP gave a result only for the smallest data set BCW in around 34 minutes and reached the time-out for all other tasks. The result for the BCW data set took ASP less than a second. ASP did also find all maximal patterns for the HD data set in approximately two hours.

Although SCIP is a well developed software we did an exemplary comparison with Gurobi [55] and CPlex [100] regarding the running times of the MILP to make sure that our results are not due to the choice of the solver. For this purpose, we chose the HD data set in the binarized version before feature selection. The data set is of manageable size for both the MILP and ASP approach but shows a large gap between the two. Using the same hardware for calculation Gurobi solved the task of finding one positive maximal pattern for the HD data set in about 23 hours and CPlex within around 8 hours. We do not want to go into detail regarding the solving heuristics of the three MIP-solvers. Here we only want to focus on the fact that all of them took several hours to solve the problem while clingo was able to present an answer within seconds. Hence, we do not believe that the gap in the calculation times between the MILP and ASP approach depends on the MIP-solver used.

The empirical study clearly shows the advantage of using ASP in the context of pattern generation in LAD with regard to calculation time. Our ASP program was faster than the MILP in all cases of finding one or all maximal positive patterns. Not only was ASP faster in calculating the patterns but in some of the cases the MILP did not even find an answer at all within our time-out of 72 hours where ASP could still finish the task. The results of this comparison show that the ASP framework is perfectly suitable for the LAD methodology.

3.5 Discussion and perspectives

In this chapter, we introduced our software `AnswerSetLAD` [14]. It is the first package for data analysis according to LAD making use of the ASP framework.

At the beginning we formulated our goals for this project. Our main goal was to provide a software that includes all LAD functionalities to allow the user to evaluate and interpret his data based on the LAD method. We perfectly achieved this goal. Especially regarding the pattern generation step we provide all commonly defined pattern types for a broad range of possible uses. `AnswerSetLAD` contains all steps needed in the LAD workflow from the original non-binary data set to the formation of a theory, which can be used for prediction of new observations.

Besides the accessibility of the LAD functionalities for a user, another reason for the development of `AnswerSetLAD` was that the various existing implementations of the method [77, 21, 70] are no longer maintained and mostly hard to extend due to untransparent encodings. The ASP environment is a natural choice if one wants to allow the comprehensibility of the code. The ASP programs for pattern generation, which we explained in detail in the former sections, are short and succinct, which is characteristic for ASP. While we are aware that ASP is not as commonly used as MILP and future developers might first have to read in the syntax and semantics of ASP and logic programming in general, we are certain that our programs are not only easy to use but also understandable and extendible with little effort. All encodings are available under [14].

The succinct language of ASP was, however, not the only cause for us to choose this framework for our software. The nature of the problems, namely enumeration of patterns, optimization of preferences and handling of Boolean functions makes ASP the perfect environment. In Section 3.4.1 we ran a comparative study between the ASP implementation of maximal pattern generation in `AnswerSetLAD` and the state-of-the art MILP approach [93]. We clearly showed that the ASP approach is by far superior to MILP regarding this problem. We are, therefore, confident that ASP and our software `AnswerSetLAD` can outperform the commonly used MILP formulations.

Future work The described comparative study between MILP and ASP is a first step. Here we compared the running times for maximal pattern generation. Even though this MILP forms the basis for most pattern calculations there are a lot more pattern types for which the performance should be investigated to make ASP commonly accepted in this field.

Regarding the pattern generation step we already mentioned that all types of patterns belonging to the LAD method are implemented within `AnswerSetLAD`. Nevertheless, we think that it can be rewarding to revise the encodings for better performance and readability. We had an insight into the use of `asprin` with respect to preference handling. Within our comparison we could not clearly decide for a strategy, with or without `asprin`, which one should use for the calculation of patterns for a data set. More tests regarding this topic should be made such that we can give a recommendation which of the two approaches should be used based on the characteristics of the data set and pattern types.

The theory formation step leaves room for further work. Here various theory types can still be added to extend the software. At this point, we refer the reader to Chapter 4 Section 4.2 were we talk more about LAD theories on a theoretical level and introduce the concept of a new theory type.

Another goal should be to implement a theory formation step in ASP without using the full pattern set as input. This approach is computationally very costly and, therefore, not useful for larger data sets.

Summarizing the results, we achieved our goals regarding the development of our software package. With `AnswerSetLAD` we provide a useful and efficient toolbox for data analysis. We are convinced that it offers a lot of possible uses for application in the biomedical as well as the pure analytical context, by offering the broad range of LAD functionalities and making them accessible to the user. Likewise, we believe that our work is an interesting innovation for the LAD community itself, as the use of ASP leads to an excellent performance with respect to the LAD requirements.

Part III

Theory

4 Theoretical extensions of LAD

This chapter presents my work on extending the theory of the LAD method. Most of the thoughts filling this part of my thesis concern the concept of prime patterns. We have seen before that prime patterns play a key role in LAD because they are succinct and easy to interpret.

This chapter is divided into two sections. In Section 4.1 we propose and discuss a new algorithm called `PrimePatternForest` for a fast generation of prime patterns in the case that the positive and negative observations have small Hamming distance.

In Section 4.2 we talk about theories built out of prime patterns, so called *prime theories*. We suggest a statistical measure that can be used to rank prime patterns and, based on this, select those prime patterns that are more significant than others to form a theory.

4.1 `PrimePatternForest` - An algorithm for the generation of prime patterns

In this section, we introduce a new algorithm that allows calculating prime patterns efficiently given a data set having small Hamming distance between positive and negative observations.

Definition 4.1. *For two binary vectors $y = (y_1, \ldots, y_n), z = (z_1, \ldots, z_n) \in \mathbb{B}^n$ the* Hamming distance *$Hamm(y, z)$ between y and z is the number of positions $i \in \{1, \ldots, n\}$ where $y_i \neq z_i$. For two sets of vectors Y and Z the* Hamming distance *$Hamm(Y, Z)$ between the sets Y and Z is the maximal Hamming distance $Hamm(y, z)$ between a pair of vectors $y \in Y$ and $z \in Z$.*

4.1.1 Basic idea

For reasons of clarity, we focus on positive patterns only. Note that each of the statements holds for negative patterns simultaneously.

By definition a positive pattern P^+ covers at least one positive observation and none of the negative observations. For a data set $\Omega = \Omega^+ \cup \Omega^-$ consisting of a disjoint union of a set of positive observations Ω^+ and a set of negative observations Ω^-, we investigate each positive observation $O^+ \in \Omega^+$ one by one generating positive patterns that cover O^+. We note here that this is possible because any term P^+ that is a positive pattern for a data set $\{O^+\} \cup \Omega^-$, for one positive observation O^+ and a set of negative observations Ω^-, remains a positive pattern for the data set if we add any new positive observation. The term P^+ might in that case cover the new positive observation or not cover it, which does not change anything about the fact that P^+ is a positive pattern. We further notice that we might end up generating the same positive pattern several times in the case that it covers multiple positive observations.

For each positive observation O^+ we consider each negative observation $O^- \in \Omega^-$ one by one to ensure that a positive pattern candidate does not cover O^-. Let us assume we have a data set consisting of only one positive observation O^+ and one negative observation O^-. We now ask the question which literals come into consideration for a positive *prime* pattern P^+ that covers O^+ and does not cover O^-. The answer is that the set of literal candidates for prime pattern generation is the set of literals that separate O^+ from O^-. Since a positive prime pattern has to cover the observation O^+, all literals of the pattern have to cover O^+. At the same time, the pattern must not cover observation O^-. We note that for a general pattern it is possible to include literals that cover observation O^-, but the property *prime* requires that the pattern has an inclusion-wise minimal set of literals. Any literal covering O^- and O^+ at the same time is redundant and could be removed such that the remaining term is still a positive pattern.

One more important remark, which we state and prove formally in Lemma 4.1, helps for the understanding of the algorithm. Assume P^+ is a positive prime pattern for a data set $\{O^+\} \cup \Omega^-$, where O^+ is one positive observation and Ω^- is a set of negative observations. We add a new negative observation O^-, $O^- \notin \Omega^-$, to the data set. Now we want to transform P^+ into a positive prime pattern for $\{O^+\} \cup \Omega^- \cup \{O^-\}$ by appending more literals to the conjunction. There are two cases that might occur. The first one is that P^+ does not cover O^-. In that case P^+ is a positive prime pattern for $\{O^+\} \cup \Omega^- \cup \{O^-\}$. The second case is that P^+ does cover O^-. In this case two scenarios are possible. The first one is that it is not possible to transform P^+ into a positive prime pattern for the new set by the conjunction with any new literal. The second one is that we need to append exactly one new literal to P^+ that discriminates O^- from O^+. This is true because of the following reasons. P^+ covers O^-. To change this situation and ensure that the transformed P^+ still covers

O^+ we need to choose at least one new literal L that covers O^+ but not O^-. The pattern $P^+ \wedge L$ still covers O^+ but does not cover O^-. We need to choose at most one such literal L, because of the inclusion-minimality of the prime pattern. Any literals appended on top of L would be redundant. We note that any new negative observation O^- cannot lead to a positive prime pattern $P' \subsetneqq P^+$, where we mean literal-inclusion.

The *set of differences* between a positive observation O^+ and a negative observation O^- is a central concept for the algorithm.

Definition 4.2 (Set of differences). *Let $\Omega = \Omega^+ \cup \Omega^- \subset \{0,1\}^n$, $n \in \mathbb{N}$, be the disjoint union of positive and negative observations, where each observation is described by n Boolean variables x_i, $i = 1, \dots, n$. Further let $O^+ \in \Omega^+$ be one positive observation and $O^- \in \Omega^-$ one negative observation. Let a_i denote the binary value that the i-th Boolean variable x_i takes in observation O^+ and b_i the binary value that x_i takes in O^-, respectively.*
The set of differences D_{O^+,O^-} *is a set of literals L_i, $i = 1, \dots, m$, $m \in \mathbb{N}$, $m \leq n$. For each Boolean variable x_i both x_i and its negation $\overline{x_i}$ are literals. If $a_i \neq b_i$, $i \in 1 \dots, n$, then*

$$\begin{aligned} L_i = x_i \in D_{O^+,O^-}, \quad & \textit{if } a_i = 1; \\ L_i = \overline{x_i} \in D_{O^+,O^-}, \quad & \textit{if } a_i = 0. \end{aligned}$$

Definition 4.3 (Positive prime pattern forest). *Let $\Omega = \Omega^+ \cup \Omega^- \subset \{0,1\}^n$ be a disjoint union of positive and negative observations.*
We consider the data set consisting of one positive observation O^+ and a set of negative observations $\Theta^- \subseteq \Omega^-$, where Θ^- is built iteratively by adding, one by one, a new negative observation $O^- \in \Omega^-$ to Θ^-. For one positive observation $O^+ \in \Omega^+$ and a set of negative observations $\Theta^- \subseteq \Omega^-$ a positive prime pattern forest *T is the union of k, $k \in \mathbb{N}$, directed trees $T_1, \dots, T_k$, each having one root r_i and each a set of leaves $l_{i1}, \dots, l_{im_i}$, $m_i \in \mathbb{N}$, $i \in \{1, \dots, k\}$. The union B of all leaves of T represents the set of positive prime patterns of the data set $\{O^+\} \cup \Theta^-$. The nodes of T are terms. There exists a directed edge from a node e to a node f in T if and only if the term $e \subseteq f$ and $|e| + 1 = |f|$, where $|\cdot|$ is the number of literals included in the term.*

Note that depending on the order in which the negative observations are considered, the resulting positive prime pattern forest for $\{O^+\} \cup \Omega^-$ differs.

The algorithm `PrimePatternForest` for the calculation of all positive prime patterns for a data set is shown in Algorithm 4. We remark that the algorithm can be used symmetrically for the calculation of all negative prime patterns.

Algorithm 4 `PrimePatternForest` - An algorithm for the generation of all positive prime patterns for a given data set.

```
input:
    Data set $\Omega = \Omega^+ \cup \Omega^-$;

output:
    Set of leaves $B \hat{=}$ All positive prime patterns

for each positive observation $O_i^+$ do
    for each negative observation $O_j^-$ do
        calculate the set of differences $D_{O_i^+, O_j^-}$;
        use each literal in $D_{O_i^+, O_j^-}$ as root of a tree in the prime pattern forest;
        delete $O_j^-$ from the set of negative observations;
    end for
    while $\exists$ a new negative observation $O_k^-$ do
        calculate the set of differences $D_{O_i^+, O_k^-}$;
        for each leaf $l$ in $B$ do
            if $l \cap D_{O_i^+, O_k^-} \neq \emptyset$ then
                return;
            else
                delete $l$ from the set of leaves $B$;
                while $D_{O_i^+, O_k^-} \neq \emptyset$ do
                    for each literal $x$ in $D_{O_i^+, O_k^-}$ do
                        if $x \in \{$ leaves in $B$ of degree $1\}$ then
                            delete $x$ from $D_{O_i^+, O_k^-}$;
                        else
                            new prime pattern candidate $P = x \wedge l$
                            while $P \neq \emptyset$ do
                                for each literal $m$ in $P$ do
                                    if $P \setminus m$ in set of leaves $B$ then
                                        delete $x$ from $D_{O_i^+, O_k^-}$;
                                    else
                                        delete $m$ from $P$;
                                    end if
                                end for
                            end while
                            add $P = x \wedge l$ to $B$ as a new leaf;
                        end if
                    end for
                end while
            end if
        end for
        delete $O_k^-$ from the set of negative observations;
    end while
end for
delete all repetitions in $B$
```

4.1.2 Example

For the illustration of the algorithm we investigate a small example. In Table 4.1 a data set of size 7×6 is shown. We use the algorithm `PrimePatternForest` to calculate all positive prime patterns of this data set. In Figure 4.1 the calculation steps of the algorithm and the resulting positive prime patterns are visualized.

We start with the positive observation 1 and negative observation 5. The set of differences is $D_{O_1,O_5} = \{\overline{x_2}, \overline{x_4}\}$. Each of these literals is used as a root of a tree in the forest as they are positive prime patterns with regard to the set $\{O_1\} \cup \{O_5\}$. Next the negative observation O_6 is added to the data set. The set of differences between O_1 and O_6 includes $\overline{x_3}$ and $\overline{x_6}$. None of these literals intersects with any of the roots/leaves of a tree in the current forest. Therefore, each conjunction can be appended as new leaves. The last negative observation O_7 yields a set of differences that intersects with each of the leaves. Therefore, the leaves stay prime and no new leaf is added.

The positive observation 2 is considered next. The set of differences of O_2 and O_5 yields five roots for trees in the prime pattern forest, namely $x_1, \overline{x_2}, \overline{x_4}, x_5$ and x_6. The positive observation 2 can be discriminated from the negative observation 6 by the literals $x_1, \overline{x_3}$ and x_5. While there exists an intersection with the roots/leaves x_1 and x_5 and they, therefore, remain prime, no intersection with the other three roots exists and the conjunction with the literal $\overline{x_3}$ can be appended. Here x_1 and x_5 cannot be used for the conjunction as they are included in the set of leaves, which means that they are prime patterns themselves at this point.

Observation 3 and observation 5 can be separated only by the literal x_5. This is the single root added for the positive observation 3. The set of differences of observation 3 and observation 6 $D_{O_3,O_6} = \{x_2, \overline{x_3}, x_4, x_5, \overline{x_6}\}$ intersects with this root and, therefore, no new leaf is added with the negative observation 6. All four literals included in the set of differences of O_3 and O_7 do not intersect with x_5. As a result all the conjunctions of x_5 with the literals of $D_{O_3,O_7} = \{\overline{x_1}, x_2, \overline{x_3}, \overline{x_6}\}$ are appended as leaves.

The last positive observation 4 of the data set differs in $\overline{x_2}, x_3$ and x_6 from observation 5. All of these literals are roots of a tree in the positive prime pattern forest. The negative observation 6 is different from observation 4 only in one attribute, namely attribute 4. Observation 4 is covered by the positive literal x_4, which is in the set of differences. The conjunction of x_4 with each of the roots leads to three prime patterns of degree 2. The set of differences $D_{O_4,O_7} = \{\overline{x_1}, \overline{x_5}\}$ again has no intersection with the existing leaves and, therefore, we get six positive prime

	x_1	x_2	x_3	x_4	x_5	x_6	
1	0	0	0	0	0	0	Ω^+
2	1	0	0	0	1	1	
3	0	1	0	1	1	0	
4	0	0	1	1	0	1	
5	0	1	0	1	0	0	Ω^-
6	0	0	1	0	0	1	
7	1	0	1	1	1	1	

Tab. 4.1.: An example data set for the illustration of the algorithm `PrimePatternForest`.

patterns from the conjunction of each of the leaves with each of the literals that cover the positive observation O_4.

After deleting the repetitions in the leaves of the positive prime pattern forest, the 18 resulting positive prime patterns of the data set are $\overline{x_2}\overline{x_3}$, $\overline{x_2}\overline{x_6}$, $\overline{x_4}\overline{x_3}$, $\overline{x_4}\overline{x_6}$, $x_1\overline{x_3}$, $x_1\overline{x_4}$, $x_5\overline{x_3}$, $x_5\overline{x_4}$, $x_6\overline{x_3}$, $x_5\overline{x_1}$, x_5x_2, $x_5\overline{x_6}$, $\overline{x_2}x_4\overline{x_1}$, $\overline{x_2}x_4\overline{x_5}$, $x_3x_4\overline{x_1}$, $x_3x_4\overline{x_5}$, $x_6x_4\overline{x_1}$ and $x_6x_4\overline{x_5}$.

4.1.3 Correctness

In this subsection, we show the correctness of the algorithm `PrimePatternForest`. Before doing so, we introduce the following Lemma, which we use in the later proof.

Lemma 4.1. *Let $\mathscr{P}$ be the set of positive prime patterns for a set of observations $\Omega = \{O^+\} \cup \Omega^-$ consisting of one positive observation O^+ and a set of negative observations Ω^-. Let $O^- \notin \Omega$ be a new negative observation. For each positive prime pattern $P^+ \in \mathscr{P}$ one of the following three cases applies:*

1. *The term P^+ is a positive prime pattern for the set $\Omega \cup \{O^-\}$.*
2. *A positive prime pattern for $\Omega \cup \{O^-\}$ can be built based on P^+ by the conjunction with exactly one literal.*
3. *The pattern P^+ cannot be transformed into a positive prime pattern for $\Omega \cup \{O^-\}$ by the conjunction with any literal or set of literals.*

Proof. A positive prime pattern has two properties. The first is that it is a positive pattern, meaning that it covers at least one positive and no negative observation. The second is that it has an inclusion-wise minimal set of literals. Let P^+ be a

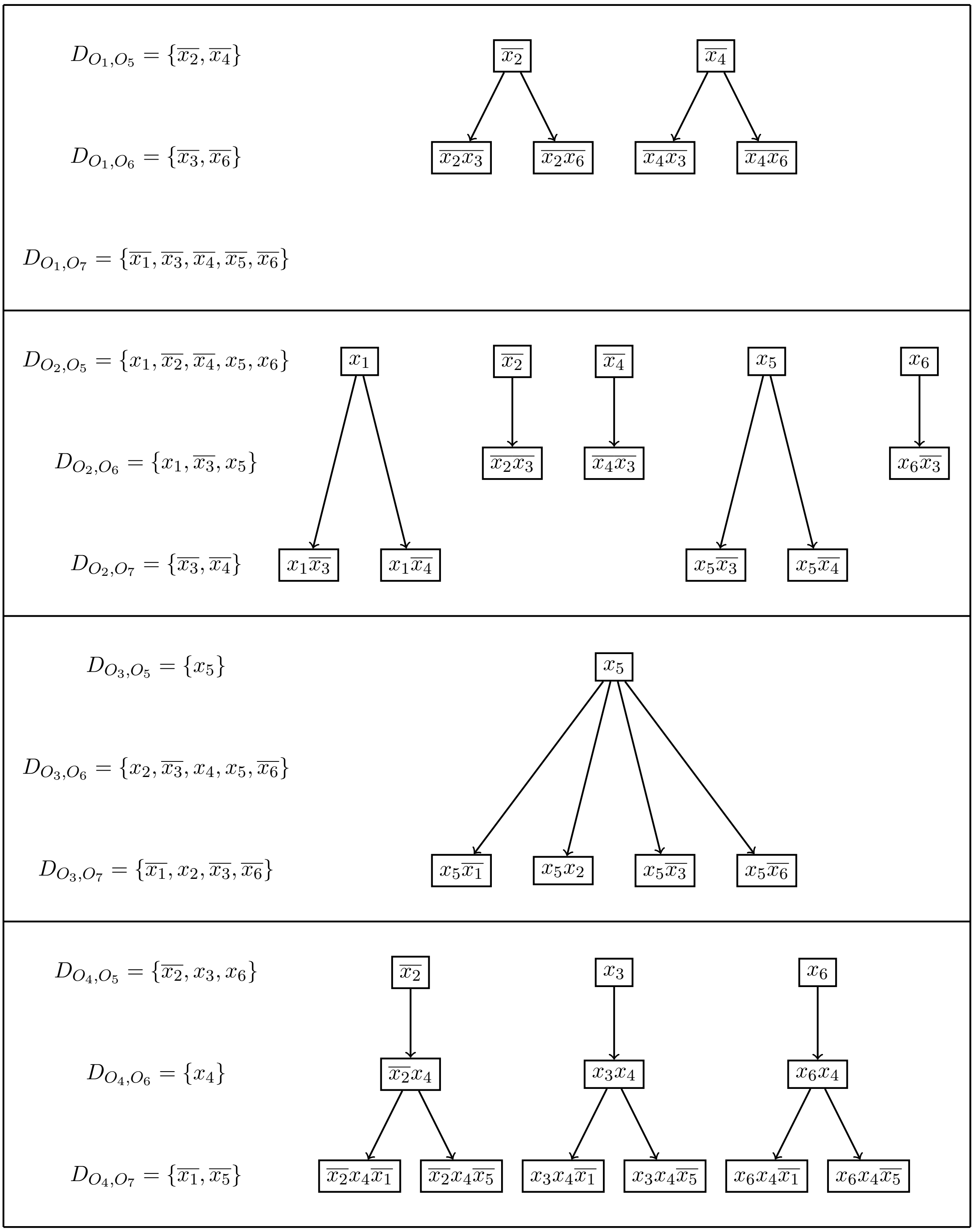

$D_{O_1,O_5} = \{\overline{x_2}, \overline{x_4}\}$
$\overline{x_2}$
$\overline{x_4}$
$D_{O_1,O_6} = \{\overline{x_3}, \overline{x_6}\}$
$\overline{x_2}\overline{x_3}$
$\overline{x_2}\overline{x_6}$
$\overline{x_4}\overline{x_3}$
$\overline{x_4}\overline{x_6}$
$D_{O_1,O_7} = \{\overline{x_1}, \overline{x_3}, \overline{x_4}, \overline{x_5}, \overline{x_6}\}$
$D_{O_2,O_5} = \{x_1, \overline{x_2}, \overline{x_4}, x_5, x_6\}$
x_1
$\overline{x_2}$
$\overline{x_4}$
x_5
x_6
$D_{O_2,O_6} = \{x_1, \overline{x_3}, x_5\}$
$\overline{x_2}\overline{x_3}$
$\overline{x_4}\overline{x_3}$
$x_6\overline{x_3}$
$D_{O_2,O_7} = \{\overline{x_3}, \overline{x_4}\}$
$x_1\overline{x_3}$
$x_1\overline{x_4}$
$x_5\overline{x_3}$
$x_5\overline{x_4}$
$D_{O_3,O_5} = \{x_5\}$
x_5
$D_{O_3,O_6} = \{x_2, \overline{x_3}, x_4, x_5, \overline{x_6}\}$
$D_{O_3,O_7} = \{\overline{x_1}, x_2, \overline{x_3}, \overline{x_6}\}$
$x_5\overline{x_1}$
x_5x_2
$x_5\overline{x_3}$
$x_5\overline{x_6}$
$D_{O_4,O_5} = \{\overline{x_2}, x_3, x_6\}$
$\overline{x_2}$
x_3
x_6
$D_{O_4,O_6} = \{x_4\}$
$\overline{x_2}x_4$
x_3x_4
x_6x_4
$D_{O_4,O_7} = \{\overline{x_1}, \overline{x_5}\}$
$\overline{x_2}x_4\overline{x_1}$
$\overline{x_2}x_4\overline{x_5}$
$x_3x_4\overline{x_1}$
$x_3x_4\overline{x_5}$
$x_6x_4\overline{x_1}$
$x_6x_4\overline{x_5}$

Fig. 4.1.: A visualization of the calculation steps of the algorithm `PrimePatternForest` including the resulting positive prime patterns of the data set given in Table 4.1.

positive prime pattern for a set of observations Ω and let $O^- \notin \Omega$ be a new negative observation.

If P^+ does not cover the new negative observation O^- then P^+ is obviously still a positive pattern for $\Omega \cup \{O^-\}$. It covers at least one positive observation, namely the same positive observation O^+ that it covers in Ω, and no negative observation, namely no negative observation in Ω and not the negative observation O^-. Also it is still prime, because it had an inclusion-wise minimal set of literals regarding Ω, which does not change by adding a new negative observation.

The more interesting scenarios arise if P^+ covers the new negative observation $O^- \notin \Omega$. In that case, P^+ is no positive pattern and, in particular, no positive prime pattern on $\Omega \cup \{O^-\}$. It is still true that P^+ covers at least one positive observation of $\Omega \cup \{O^-\} \subset \Omega$ and no negative observation in this data set apart from O^-. Let L be any fixed literal that covers the positive observation O^+, which is covered by P^+, and does not cover O^-. Such a literal must exist because otherwise $O^+ = O^-$, which contradicts the main property $\Omega^+ \cap \Omega^- = \emptyset$ of our data set. The term $P^+ \wedge L$ covers O^+ and does not cover O^- by construction. It, therefore, covers a positive observation and no negative observation in $\Omega \cup \{O^-\}$. This is true because we already noted that P^+ does not cover any negative observation apart from O^- so we can be sure that a combination of P^+ with any literal does not cover any of them. It follows that $P^+ \wedge L$ is a positive pattern for $\Omega \cup \{O^-\}$.

If the positive pattern $P^+ \wedge L$ does not include any shorter positive pattern, then the pattern is prime and we are done. This is the second case of the Lemma.

The last possibility is that $P^+ \wedge L$ is not prime on the data set $\Omega \cup \{O^-\}$. This means that the deletion of a literal of $P^+ \wedge L$ leads to a positive pattern on $\Omega \cup \{O^-\}$. Now there are two options. The first option is that there exists another literal $M \neq L$ that covers O^+ and not covers O^- and that can be used to form a new term $P^+ \wedge M$, which is a positive prime pattern on $\Omega \cup \{O^-\}$. Here we then found a positive prime pattern by adding exactly one literal. The second option is that none of the literals that cover O^+ and not cover O^- lead to a prime pattern on $\Omega \cup \{O^-\}$. In this case it is easy to see that we cannot find a positive prime pattern for $\Omega \cup \{O^-\}$ that includes P^+. To ensure that the new term does not cover O^- we have to append literals that do not cover O^-, but those literals have to cover O^+ such that we do not lose the property that the term covers this positive observation. However, we already know that each combination $P^+ \wedge K$ for any such literal K is not prime on $\Omega \cup \{O^-\}$, meaning that it includes a shorter positive pattern. Therefore, any larger combination with valid literals would include a shorter positive pattern as well. This means that we cannot transform P^+ into a positive prime pattern on $\Omega \cup \{O^-\}$. □

Lemma 4.1 immediately leads to a bound on the degree of positive prime patterns of a data set.

Proposition 4.1. *Let P^+ be a positive prime pattern for a set of observations $\Omega = \Omega^+ \cup \Omega^-$. Then for the degree $deg(P^+)$ of P^+ holds:*

$$deg(P^+) \leq |\Omega^-|.$$

The upper bound given in Proposition 4.1 is obviously very high in general as we look at much larger data sets in practice.

The following proposition, which gives a bound on the number of positive prime patterns, follows from Lemma 4.1 and the construction of the set of leaves in the Algorithm `PrimePatternForest`.

Proposition 4.2. *Let O^+ be a positive observation and D_{O^+,O_i^-}, $i = 1, \ldots, m$ the sets of differences for m negative observations $O_1^-, \ldots, O_m^-$. If the pairwise intersection of the sets of differences is empty for all negative observations $O_1^-, \ldots, O_m^-$, then the following statements hold:*

1. *The number of positive prime patterns that cover the positive observation O^+ equals $\prod_{i=1}^m |D_{O^+,O_i^-}|$.*
2. *The degree $deg(P^+) = \sum_{i=1}^m 1 = m$ for all positive prime patterns P^+ that cover O^+.*

The intuition behind the second statement in Proposition 4.2 is that if the pairwise intersection of the sets of differences is empty, then for each new negative observation exactly one literal has to be added to each leaf in the prime pattern forest. As we assume a data set with m negative observations, each of the resulting prime patterns has degree m. The first statement is based on the same argument. If the pairwise intersection of the sets of differences is empty, then all combinations of literals have to be added as new leaves to the prime pattern forest.

Following up on these general observations, we now prove the central theorem of this section.

Theorem 4.1. *The output set B of the algorithm* `PrimePatternForest` *includes exactly all positive prime patterns of the data set Ω.*

Proof. We prove the correctness of the algorithm `PrimePatternForest`. In a first step we show that each term included in the output set, namely each leaf in B, is a positive prime pattern.

Therefore, we start looking at lines 3 to 7. In this first step of the algorithm only two observations are in consideration. One positive observation O_i^+ and one negative observation O_j^-. We calculate the set of literals that separate O_i^+ from O_j^-. Obviously all these literals were prime patterns of degree 1 if only this one negative observation existed. All of them are used as roots of a tree in the prime pattern forest. At this point the output would, therefore, consist of (all) positive prime patterns if no more negative observations were in the queue.

Now in the case that more negative observations exist, we look at them one by one (see lines 9 and 10). Again the set of differences is calculated for the positive observation O_i^+ and the new negative observation O_k^-.

For each leaf l of the forest we then look for an intersection with the set of differences $D_{O_i^+,O_k^-}$. Note here that we refer to the roots as leaves too if they have no out-degree. If such an intersection exists (line 12 and 13), then the leaf remains a prime pattern (up to the set of negative observations considered), because at least one literal in the leaf is different from the new negative observation O_k^-. This means that the term considered obviously covers the positive observation and does not cover the new negative observation. For the already processed negative observations this is true recursively. The leaf l, therefore, remains a positive prime pattern for the new data set including the new negative observation O_k^-.

If there is no intersection (line 14 and following), then the leaf appears in the negative observation O_k^-. We know with Lemma 4.1 that we have to add exactly one literal x to the leaf l to make it a prime pattern covering O_i^+ with regard to the new negative observation O_k^- or there is no option to transform l into a valid prime pattern regarding O_i^+ and O_k^-. In line 17 to 18 the literals in the set of differences $D_{O_i^+,O_k^-}$ are checked for an intersection with the leaves of degree 1. If such an intersection exists, then the literal x itself is a prime candidate and cannot be added to any of the leaves.

If not we proceed. Before we can add the combination of the leaf l with literal $x \in D_{O_i^+,O_k^-}$ to the set of leaves B, we have to ensure that no subterm of $x \wedge l$ is already a leaf of the forest, meaning that this is already a prime pattern up to the given point. Therefore, we check for each literal m in $x \wedge l$ whether the subterm $(x \wedge l) \setminus m$ is included in the leaves B. If this is the case, the literal x cannot be added. In the case that none of the subterms already exists in B we can add $x \wedge l$ to B. The new leaves are prime patterns regarding all negative observations considered up to this point, because they were prime patterns up to the negative observation considered before by construction and now only one literal is added to make the term different from O_k^-. No shorter change is possible. Therefore, the leaf is a prime pattern.

To assure ourselves that we calculate *all* positive prime patterns we again look at the first negative observation. Here no other possible literals than the literals in the set of differences exist as prime patterns. At this step we, therefore, find *all* prime patterns up to the one negative observation added. As we do the same for each new negative observation, namely look at all literals that separate the negative observation from the positive observation and then add all possible combinations, we generate all prime patterns up to the current negative observation in each iteration and finally all positive prime patterns for the data set. □

4.1.4 Performance

Setup

We started this section with the statement that our algorithm `PrimePatternForest` is particularly favorable when the underlying data set has a small Hamming distance between the set of positive and negative observations. Having a look at the algorithm we see that only those literals appearing in a set of differences D_{O^+,O^-} are taken into account in each iteration step. As we have $Hamm(O^+, O^-) = |D_{O^+,O^-}|$ our hypothesis is reasonable, because we use less calculation steps the closer the two observations O^+ and O^- are.

To substantiate this observation we did a comparison of the running time of a `python` implementation of `PrimePatternForest` on randomly generated data sets of fixed size and differing maximal Hamming distance between positive and negative observations. We, therefore, calculated binary matrices of size 10×10 having a maximal Hamming distance between positive and negative observations of 3, 6 and 9 and of size 20×20 having a maximal Hamming distance between positive and negative observations of 5, 10 and 15. For each of the six properties, we generated three binary data sets. We then used the algorithm `PrimePatternForest` to calculate all positive prime patterns for each of the data sets.

Additionally, we compared our results to the running time of our `python` implementation of the classical Term Enumeration Algorithm (see Chapter 2 Algorithm 1) proposed by [24] and the running time of our software `AnswerSetLAD`. The results are shown in Table 4.2. The running times are the averaged results over the three data sets for each of the data set properties. All times are CPU times of calculations on a Linux AMD64 with 3.20GHz, 4 cores and 15.6GB of memory.

Size	10×10	10×10	10×10
$Hamm(\Omega^+, \Omega^-)$	3	6	9
PrimePatternForest	0.68s	0.94s	3.69s
Term Enumeration	19m12s	26m42s	12m31s
AnswerSetLAD	0.02s	0.03s	0.05s

Size	20×20	20×20	20×20
$Hamm(\Omega^+, \Omega^-)$	5	10	15
PrimePatternForest	28.2s	38m23.4s	57m7.7s
Term Enumeration	($\geq$ 12 hours)	($\geq$ 12 hours)	($\geq$ 12 hours)
AnswerSetLAD	0.8s	12.5s	33.7s

Tab. 4.2.: Averaged results of the running time (CPU time) of the algorithms PrimePatternForest and Term Enumeration (see Algorithm 1 in Chapter 2) and the software AnswerSetLAD on randomly generated binary data set having a fixed maximal Hamming distance between positive and negative observations. Each time shown is the average of the running time over three data sets.

Results

The results in Table 4.2 obtained from calculations on data sets of size 10×10 for the algorithm PrimePatternForest show that the problem gets more difficult with a higher maximal Hamming distance. In contrast, we see that the running time of the Term Enumeration algorithm does not depend on the maximal Hamming distance used for the generation of the data sets. It needs the largest averaged running time for data sets having maximal Hamming distance 6, followed by those having maximal Hamming distance 3 and shows the fastest averaged running time for data sets having maximal Hamming distance 9. In all three cases the PrimePatternForest algorithm is much faster than the Term Enumeration method. PrimePatternForest was able to solve all problems with an averaged running time under four seconds while the Term Enumeration method used between ten and thirty minutes. These results suggest that PrimePatternForest is a valuable alternative to the classical Term Enumeration approach. Although PrimePatternForest is definitely preferable to Term Enumeration, the comparison to AnswerSetLAD shows that our ASP tool clearly outperforms the presented algorithm. It solved all three problems with running times under 0.1 seconds.

The second table, which includes running times for the data sets of size 20×20, supports the observations made before. PrimePatternForest is able to solve all given problems of size 20×20 in times starting from a few seconds for the data sets with smaller maximal Hamming distance and going up to one hour for the data sets with higher maximal Hamming distance. We again see a correlation between running time and maximal Hamming distance. The Term Enumeration method was not able to calculate the positive prime patterns for any of these bigger data sets within our time-out of 8 hours. Still we see that `AnswerSetLAD` is definitely preferable in all cases but especially when the Hamming distance of the data set is large.

Concluding from this performance study we summarize that `PrimePatternForest` scales well on data sets having a small Hamming distance between the set of positive and negative observations. It is clearly preferable to the classical Term Enumeration algorithm formulated in [24] on all tested data sets. Nevertheless, our ASP software `AnswerSetLAD` is by far unbeaten in running time.

4.1.5 Discussion

In this section, we introduced a new algorithm for the calculation of prime patterns. We saw that it is a meaningful alternative to the Term Enumeration algorithm, which was proposed in [24], because it solves problem instances noticeably faster. Although the software `AnswerSetLAD` leads to better results regarding running time in all our experiments, we still think that the algorithm `PrimePatternForest` follows an interesting idea about the composition of prime patterns.

The idea for the algorithm arose from a biological application including a data set from perturbation measurements on a signaling network in a cell (see Chapter 5). Perturbation measurements are measurements of activation of proteins in the cell network under different perturbed settings. As these settings vary only slightly the observations tend to be similar to each other, meaning that the measurements differ in only a few proteins. Here the `PrimePatternForest` algorithm could find a suitable application. We think that it would be interesting to have a closer look on biological data sets to find a niche where our algorithm can be of value. As it can be assumed that biological data for a system is not independent from each other, we are confident that such an application exists.

4.2 Prime theories and core theories

In this section, we present ideas and preliminary results on LAD theories. We put the focus on theories formed by prime patterns. Those theories are called *prime theories*. We thereby follow the definition of Yves Crama, Peter L. Hammer and Toshihide Ibaraki [34].

Definition 4.4. *Let ϕ be a DNF consisting only of prime patterns of a pdBf (Ω^+, Ω^-). Then ϕ is called a* prime theory *for (Ω^+, Ω^-).*

Prime theories play a central role in LAD. On the one hand, this is due to their structure built out of prime patterns, which are succinct and more easy to interpret than longer patterns. On the other hand, it is convenient in practice that for each pdBf we are able to find a prime theory. This is true, because for each positive (negative) observation, we find a positive (negative) prime pattern that covers the observation. This prime pattern is the characteristic term of the observation itself in the "worst" case when all other literal combinations cover an observation of the opposite sign or a shorter pattern covering the observation. A disjunction of any set of patterns that covers all observations is a theory.

Obviously the number of prime theories for a data set is large. In this section, we investigate the question of how to choose a reasonable prime theory out of the pool of possible ones. More precisely, we suggest a statistical measure that can be used to rank the prime patterns for a data set according to their literals and leads to a subset of prime theories, which we will refer to as *core theories* in the following.

4.2.1 Core theories

As mentioned before any prime theory for a pdBf (Ω^+, Ω^-) is a disjunction of some of its prime patterns. The number of prime patterns for a pdBf is large in general and, therefore, the calculation of the full set of prime patterns is a difficult and intensively investigated task [24, 61, 83]. Nevertheless, for data sets like the six examples we introduced before (see Chapter 3 Table 3.1) from the UC Irvine Machine Learning Repository [38] having less than 1000 observations and less than 1500 attributes, we are able to calculate the full set of prime patterns in short time using our software package `AnswerSetLAD`. Based on that we do not worry about how to calculate the full set of prime patterns here, but start from the point were we already have it at hand. Regarding the problem of choosing a *good* theory out of the set of all possible prime theories for a data set, we are interested in the question: How can we determine prime theories that yield better results regarding their prediction accuracy than others?

For this purpose, we have a closer look at the set of prime patterns belonging to a data set. More precisely we investigate the occurrence of their literals over the whole set. To illustrate our approach we consider the discretized BLD data set after feature selection. There are 16 positive and 48 negative prime patterns in the data. Our basic idea is that literals that appear more frequently within the whole set of prime patterns are more trustworthy than literals that appear only a few times. In Figure 4.2 we see a visualization of the set of prime patterns of the BLD data set. Here the prime patterns are divided into their literals and then the number of times each literal appears in the whole set of prime patterns is shown.

All literals that exist in the data set are listed on the X-axis. Each occurrence of a literal in a positive prime pattern is visualized using a blue square. Each occurrence of a literal in a negative prime pattern is visualized using a red square.

To choose a prime pattern from the set of all prime patterns we define the *core weight*:

Definition 4.5. *For a pdBf* (Ω^+, Ω^-)*, a literal* L *and its complement* $\overline{L}$*, let*

- $\Theta_+(L)$ *be the number of occurrences of* L *in a positive prime pattern of* (Ω^+, Ω^-)*;*
- $\Theta_-(L)$ *be the number of occurrences of* L *in a negative prime pattern of* (Ω^+, Ω^-).

The positive core weight $cw^+(L)$ *of the literal* L *is defined by:*

$$cw^+(L) := \frac{\Theta_+(L)}{\Theta_+(L) + \Theta_+(\overline{L})} + \frac{\Theta_-(\overline{L})}{\Theta_-(\overline{L}) + \Theta_-(L)}. \tag{4.1}$$

The negative core weight $cw^-(L)$ *of a literal* L *is defined by:*

$$cw^-(L) := \frac{\Theta_-(L)}{\Theta_-(L) + \Theta_-(\overline{L})} + \frac{\Theta_+(\overline{L})}{\Theta_+(\overline{L}) + \Theta_+(L)}, \tag{4.2}$$

The core weight *of a positive (negative) prime pattern is the sum of the positive (negative) core weights of its literals. The* core weight *of a theory is the sum of the core weights of its positive prime patterns and the core weights of its negative prime patterns.*

Thus the positive core weight of a literal L is the number of its occurrences in a positive prime pattern normalized by the total number of times the variable occurs over all positive prime patterns plus the number of times the complementary literal

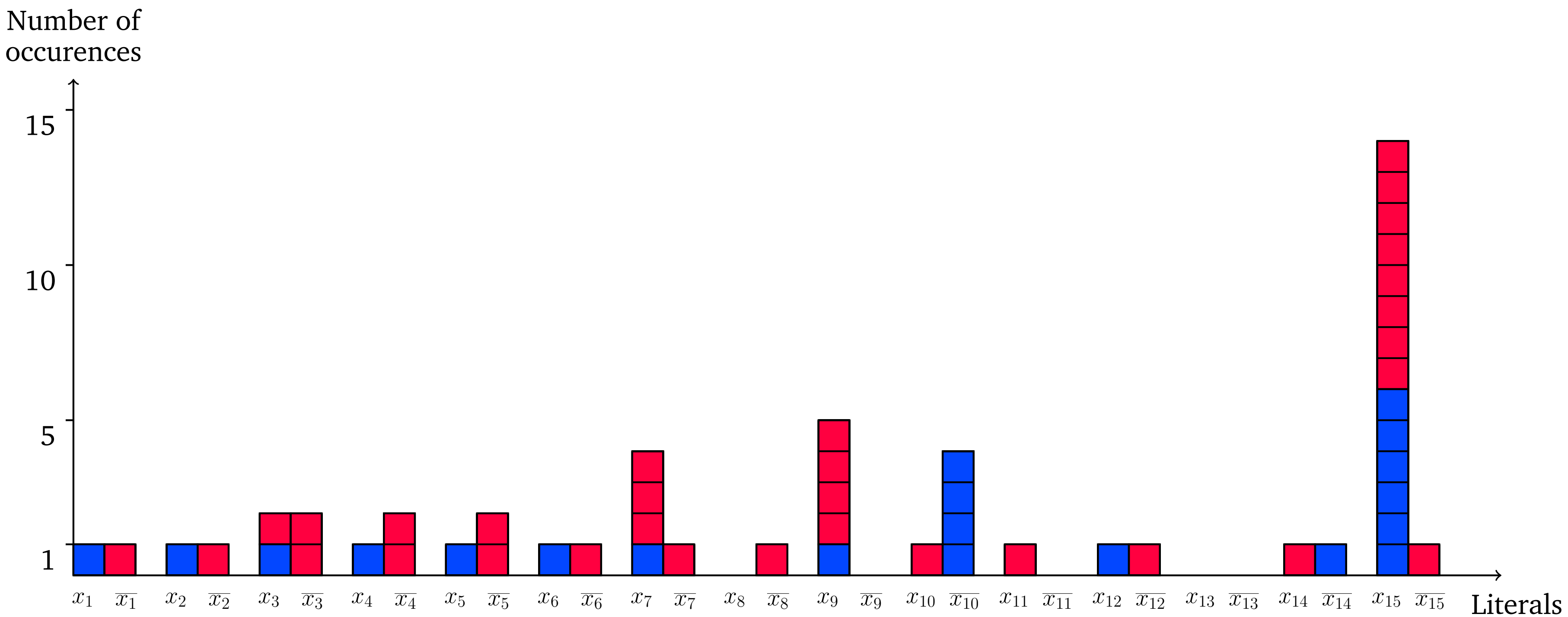


Fig. 4.2.: The occurrences of the literals in the positive and negative prime patterns of the BLD data set after feature selection. Each occurrence of a literal in a positive prime pattern is visualized using a blue square. Each occurrence of a literal in a negative prime pattern is visualized using a red square.

$\overline{L}$ occurs in the set of negative prime patterns normalized by the total number of times the variable occurs over all negative prime patterns.

The ideas behind the definition of the core weight for the choice of a positive prime pattern are:

- If a literal L occurs frequently in the set of positive prime patterns, then it is more reliable than a literal that occurs less often.
- If the complement of L, the literal $\overline{L}$, occurs frequently in the set of negative prime patterns, then the literal L is more reliable for a positive prime pattern.
- A positive prime pattern is more reasonable to use in a theory, when it consists of a set of reliable literals.

According to these ideas and our definition of core weights, we aim to choose those positive prime patterns to form a theory that consist of literals that appear frequently over the set of positive prime patterns and whose complements appear frequently over the set of negative prime patterns. The same ideas hold for the choice of negative prime patterns. A theory that has a high core weight is more reasonable than a theory that has low core weight.

Based on the definition of core weights of prime patterns we can now define a subclass of prime theories, which we call *core theories*.

Definition 4.6. *Given a pdBf* (Ω^+, Ω^-)*, a prime theory* ϕ *that has maximal core weight is called a* core theory.

4.2.2 Implementation

For the implementation of our described ideas about prime and core theories we used ASP to integrate the calculation into the software package `AnswerSetLAD`.

Input: The set of prime patterns

To generate the set of all (prime) patterns, or a subset of it, in a convenient way, we used our `python` script `AllPatterns.py` that allows enumerating all patterns of any type within our toolbox `AnswerSetLAD` [14]. The parameters *lower bound on the degree*, *upper bound on the degree*, *lower bound on homogeneity* and *lower bound on prevalence* can be added to specify the desired set of patterns. The resulting `txt`-file is then transformed into an ASP input file for our theory calculation by the parser

ReadAllPatternsForCoverCalc.py. The input file includes for each prime pattern for each of its literals the predicate pat(Id,Sign,Degree,(Variable,Value)) and for each of the observations that is covered by the pattern the predicate cov(Id,Sign,Degree,(Sign,ObsId)). A pattern can uniquely be identified by the combination of the first three entries of the predicates. Patterns of different sign or degree can have the same ID. An example is given below:

```
pat(2,1,3,(19,1)).
pat(2,1,3,(23,0)).
pat(2,1,3,(7,0)).
cov(2,1,3,(1,10)).
cov(2,1,3,(1,15)).
```

This pattern has the ID number 2. It has sign 1 , which means that it is a positive pattern, and has degree 3. Its literals are variable 19 with a positive value, 23 with a negative value and 7 with a negative value. The pattern covers the positive observations 10 and 15.

Calculating a minimal prime theory

We implemented an ASP program to choose a minimal prime cover from the set of all prime patterns of a data set calculated and given in the form described above. We remind that a pattern cover is a set of patterns for a data set, such that all observations are covered by at least one pattern (see Chapter 2 Section 2.4.2). A prime cover consists of prime patterns only. A pattern cover is called minimal, if it includes a minimal number of patterns. Besides the prime patterns, the algorithm needs the data set in suitable format (see Chapter 3 Figure 3.3) as input. The encoding is shown in Figure 4.3.

The code follows the generate-define-test structure often used in ASP programs. In line 2 a set of patterns primecover is chosen from the set of prime patterns pat. As explained before, a pattern can be uniquely identified by the three variables N,S,D standing for ID, sign and degree, respectively. Up to that point any set of prime patterns could be chosen. To narrow down the answer set we use the following constraints. In line 5 a predicate iscovered is defined that is true for each observation P of sign X that is covered by a pattern included in primecover. To get an answer set that represents a full cover of the data set, it is not allowed that any of the observations remains uncovered. This is the statement given in the integrity constraint in line 8. We minimize the number of patterns in the cover (line 11) to

```
% GENERATE
{ primecover(N,S,D) } :- pat(N,S,D,_).

% DEFINE
iscovered(P,X) :- primecover(N,S,D), cov(N,S,D,(P,X)).

% TEST
:- i(P,X,_,_), not iscovered(P,X).

% OPTIMIZE
#minimize{ 1,N,S,D : primecover(N,S,D) }.
```

Fig. 4.3.: The encoding for the generation of a minimal prime cover from the set of prime patterns.

get a minimal prime cover. A prime cover resulting from this calculation is called *minimal prime cover* in the following.

Calculating a minimal core theory

We introduced the concept of core theories as a subset of the prime theories for a given pdBf (Ω^+, Ω^-). Here we explain our implementation in ASP. The full encoding is shown in Figure 4.4.

Similar to the program for the calculation of minimal prime theories, this encoding is based on the full set of prime patterns of a data set. It is organized using the generate-define-test structure. While the generate and test parts coincide with those of the minimal prime theory calculation, the define part is longer as the predicates for the core weights need to be defined. The predicate `occ(((Y,V),S),Q)` saves for each literal consisting of variable `Y` and value `V` the occurrence `Q` in the set of positive prime patterns (`S` $= 1$) and negative prime patterns (`S` $= 0$). Therefore, in line 8 for each literal that appears in a pattern the number of times it occurs is counted and saved in `Q`. Because some literals do not occur at all in a pattern of sign `S`, we need lines 9 and 10 for further calculations.

In a next step the weights are assigned. In predicate `literalweight(((Y,V),S), P,Q,N,R)` for each literal `(Y,V)` the four measures `P`,`Q`,`N` and `R` are saved, where `P` stands for Θ_+ of the literal `(Y,V)`, `Q` for Θ_+ of the complement of `(Y,V)`, `N` for Θ_- and `R` for Θ_- of the complement of `(Y,V)`.

In the lines 15 and 16 the full literal weight is calculated using the definition of the core weight of a literal shown in (4.1) and (4.2). The weight `W` of literal `(Y,V)` for the

```
1  % GENERATE
2  { primecover(N,S,D) } :- pat(N,S,D,_).
3
4  % DEFINE
5  theorycov(P,X) :- primecover(N,S,D), cov(N,S,D,(P,X)).
6
7  occ(((Y,V),S),Q) :- pat(_,S,_,(Y,V)),
8  Q = #sum{1,N,S,D : pat(N,S,D,(Y,V)) }.
9  occ(((Y,V),S),0) :- pat(_,S,_,_), pat(_,_,_,(Y,V)),
10         not pat(_,S,_,(Y,V)).
11
12 literalweight(((Y,V),S), P,Q,N,R) :- occ(((Y,V),S),P),
13         occ(((Y,X),S),Q), occ(((Y,V),W),N), occ(((Y,X),W),R),
14         V!=X, W!=S.
15 fullliteralweight(((Y,V),S), W) :- literalweight(((Y,V),S), P,Q,N,R),
16         W=P*(N+R)-N*(P+Q).
17
18 patternweight((N,S,D),K) :- pat(N,S,D,_),
19         K = #sum{W, (Y,V),S  : fullliteralweight(((Y,V),S), W),
20         pat(N,S,D,(Y,V)) }.
21
22 % TEST
23 :- i(P,X,_,_), not theorycov(P,X).
24
25 % OPTIMIZE
26 #minimize{ 1@2,N,S,D : primecover(N,S,D) }.
27 #maximize{ K@1 : patternweight((N,S,D),K), primecover(N,S,D) }.
```

Fig. 4.4.: The encoding for the generation of a minimal core cover from the set of prime patterns.

patterns of sign `S` is then stored in the predicate `fullliteralweight(((Y,V),S), W)`.

In a last definition in lines 18 to 20 the weight `W` of a pattern is calculated and stored within the predicate `patternweight((N,S,D),K)`. We remind that a pattern can be uniquely identified by the tuple `(N,S,D)`, where `N` is its ID, `S` is its sign and `D` is its degree. The core weight `W` of the pattern `(N,S,D)` is the sum over all its literal weights.

The optimization part consists of two optimization statements. We first minimize the number of patterns within a theory and then maximize the core weight of the theory. The answer sets of this encoding are, therefore, a subset of the answer sets of the encoding in Figure 4.3, which calculates all prime covers of minimal size.

4.2.3 Discussion

In this section, we illustrated our ideas on how a statistical measure on the literals of all prime patterns of a data set can be defined to select more reliable patterns and with that build theories.

With the inclusion of statistics into the LAD process we take a step in a new direction, which to our knowledge has not been done before. Although measures of statistical significance have been mentioned in the context of pattern selection, see for example [59], the idea of looking at their literal occurrences is a new concept. It is important to note again that the calculation of all prime patterns is computationally very expensive and that it might not be worth the effort when data sets are big. Nevertheless, we think that the ideas explained before are not only interesting for the pattern selection process on the way to build theories but also to get an insight on how important a certain literal might be. Especially when we speak about interdisciplinary work it is useful to think of understandable pieces of information that can be communicated across different research fields. We think that measures on literals can be such pieces of information.

In this subsection, we showed preliminary ideas and implementations. In future work these concepts should be tested within an extensive study. The comparison of all prime theories for a data set is, however, computationally very expensive and time consuming. This is due to the fact that the number of prime theories for a data set is rapidly growing with its size. For that reason, we leave this as a perspective.

Furthermore, different ways of weighting the patterns according to their literal occurrences could be investigated in subsequent work. The way we set up our workflow, we are very flexible to adjust the procedure according to new ideas. We think that it could be of interest to not only investigate single literals but stick more closely to the original LAD method and look for *subpatterns* in the set of patterns. When we generate all prime patterns for a data set we see that often patterns for a fixed degree n are formed out of building blocks of size $n-1$ or less, combined with one or a few changing literals. We believe that those building blocks can be expanded to an interesting and informative concept.

The goal to find a good theory out of all possible theories is central for the LAD methodology. As we speak about partially defined Boolean functions, *known* and *unknown* data, there will obviously never be a method that can decide for a theory that will not make errors. The unknown data is *unknown* and we cannot change that fact. We can, nevertheless, try to constrain our theories to the ones which seem to be more promising according to certain criteria. Here we showed one idea regarding this important task.

Part IV

Application

Biological applications 5

5.1 Perturbation data of signaling networks

This first application of our methods was published as a workshop paper with Martin Gebser, Torsten Schaub and Alexander Bockmayr [15]. Based on a cooperation with the group of Nils Blüthgen at Humboldt University of Berlin and their work on perturbation experiments [64, 37], I had the idea of analyzing signaling networks via Logical Analysis of Data. In this section, we show exemplarily on the EGFR signaling pathway how LAD and our software `AnswerSetLAD` can be used to identify protein interactions.

5.1.1 Biological background

Regulatory signaling networks The regulatory signaling network of a cell consists of several molecular regulators interacting with each other to coordinate cell actions. Errors in these networks are responsible for diseases such as cancer. Systems biology, therefore, investigates cell signaling to understand how changes in the network structure affect the flow of information. The final goal is to target certain proteins of the network to stop the mutation or deregulation of a diseased cell or lead it to apoptosis.

Targeted therapy Mutations happen every time that cells divide. Normally, the immune system can cope with it. But when the number of mutations increases or the immune system is stressed, some mutated cells might not be spotted and can grow without being eliminated. Cancer therapy is a huge research field. Nowadays chemotherapy is still a method of choice. It works by cutting the signaling pathways of the cell to stop cell division. This is useful to stop the cells from growing, but it is not selective enough for cancer cells. For that reason, chemotherapy leads to numerous undesirable side effects. Targeted therapy in contrast aims to affect only specific cells and not interfere with the healthy parts of the body. A precise understanding of the signaling networks in cells is fundamental for this approach.

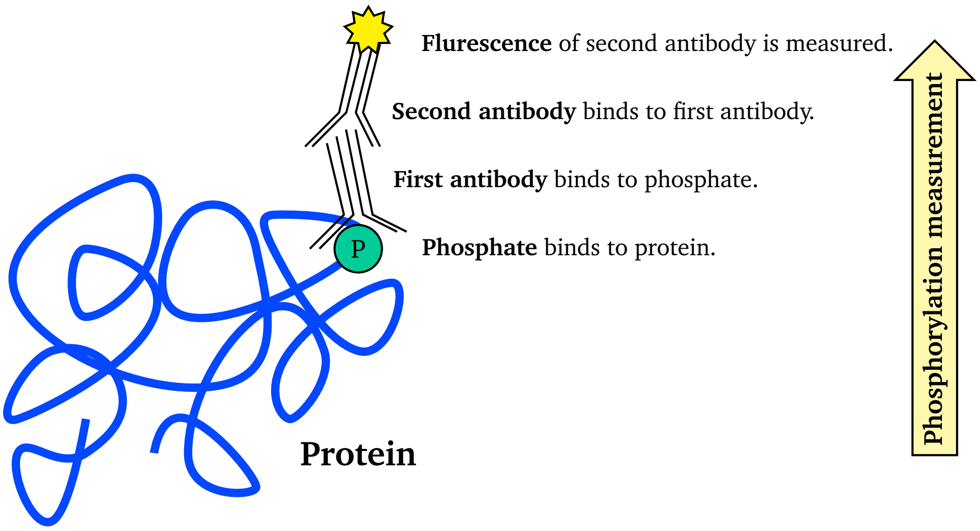

Flurescence of second antibody is measured.
Second antibody binds to first antibody.
First antibody binds to phosphate.
Phosphate binds to protein.
P
Protein
Phosphorylation measurement

Fig. 5.1.: An intuition of the phosphorylation of a protein. A phosphorylation measurement answers the question of how many phosphates bind to the protein. A protein can bind different phosphates. Different phosphates can activate different pathways.

Phosphorylation Phosphorylation measurements are a commonly used approach for the study of regulatory signaling networks. The number of phosphates that bind to a protein of the cell is counted under varying conditions (see Figure 5.1 for an intuition). The phosphorylation measurements are then used to indicate that a protein was active or inactive under the specific setting.

The Epidermal Growth Factor Receptor (EGFR) The *Epidermal Growth Factor Receptor* (EGFR) is a protein in the cell surface. Its phosphorylation activates several signal transduction cascades such as the MAPK and AKT pathways that are associated with cancer. It is known that an up-regulation or constant activation of EGFR due to mutations of the cell leads to uncontrolled cell division. The protein EGFR is, therefore, an interesting target for cancer studies in signaling networks.

5.1.2 Identifying protein interactions from phosphorylation measurements via pattern generation

In Table 5.1 we see discretized phosphorylation measurements of proteins in the EGFR signaling network. The original data before discretization is taken from the perturbation experiments conducted by [64]. In Figure 5.2 we show an idea of the known underlying protein network. This figure is adapted from [64].

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
S6K	TGF$_\alpha$	IGF	MEK inh	PI3K inh	MEK	AKT	ERK
1	1	0	0	0	0	0	0
1	0	1	0	0	0	1	1
1	1	0	1	0	1	1	0
1	0	1	1	0	1	1	0
1	1	0	0	1	1	0	1
0	0	0	1	0	1	0	0
0	0	0	0	1	0	0	1
0	0	1	0	1	1	1	1

Tab. 5.1.: Discretized phosphorylation measurements of the EGFR pathway. The original data before discretization is taken from the experiments made in [64].

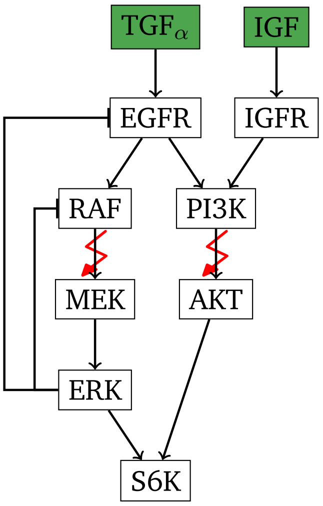


Fig. 5.2.: The EGFR-signaling pathway adapted from [64].

The perturbation experiment The phosphorylation of the proteins MEK, AKT, ERK and S6K was measured under different combinations of a stimulus at one of the growth factors TGF$_\alpha$ and IGF, and inhibitions before MEK and after PI3K. In Table 5.1 the active stimulus is represented by 1 and the absence of a stimulus is represented by 0. The same holds for the presence of an inhibition (labeled with 1) or its absence (labeled with 0). In Figure 5.2 the stimuli are marked in green and the points of inhibition are marked by a red arrow.

Preprocessing the data set We divided the data in Table 5.1 by the readout at the protein S6K that is located at the end of the pathway. This leads to a division into positive and negative observations of the data set. All observations having a high phosphorylation at S6K (indicated by 1) are positive observations and all observations having a low phosphorylation at S6K (indicated by 0) are negative observations. We note here that we could divide the table by any of the observed proteins depending on the research question.

In the following we investigate the data set looking for prime patterns that explain the outcome of the phosphorylation of the downstream protein S6K.

Positive prime patterns We divide our analysis of positive prime patterns in patterns including a stimulus at TGF$_\alpha$ and patterns including a stimulus at IGF.

The search for positive prime patterns including TGF$_\alpha$ leads to a single solution, namely `pat(1,1)`, a prime pattern of degree one. This prime pattern is the only prime pattern of degree one for this data set. The pattern suggests that using a stimulus at TGF$_\alpha$ the phosphorylation of S6K is independent from the use of additional inhibitions. When we look at the schematic overview of the known information on the network in Figure 5.2 this outcome seems obvious. The signal starting in TGF$_\alpha$ can be transmitted using both pathways over MEK or over AKT and can thus reach S6K no matter the inhibition of one of the proteins.

The positive prime patterns including a stimulus at IGF all have degree two. The answer sets are shown below:

```
Answer: 1
pat(2,1) pat(4,0)
Answer: 2
pat(2,1) pat(5,0)
Answer: 3
pat(2,1) pat(3,1)
Answer: 4
pat(2,1) pat(7,0)
```

Those stable models can nicely be interpreted in the biological context. The stimulation of IGF does not guarantee a phosphorylation at S6K as the pathway can be cut using an inhibition at PI3K. Therefore, we do not see a prime pattern of degree one including IGF.

Answer 2 illustrates the observation that we can have a positive readout at S6K while stimulating IGF without seeing a phosphorylation of MEK. Answer 4 is a similar observation including the information that we do not have a phosphorylation of ERK when we see a phosphorylation of S6K under the given stimulus. Those two stable models indicate that MEK and ERK might not lie on the pathway from IGF to S6K.

Answer 3 can be interpreted in the way that an inhibition at MEK does not prevent a phosphorylation of S6K when using the stimulus at IGF, because the signal starting at IGF is not running through MEK on the way to the downstream protein.

Answer 1 stands for the stimulation of IGF without the inhibition of PI3K. This prime pattern represents the path from IGF along PI3K. It is also the only positive prime pattern including IGF having a higher prevalence than the other three patterns.

Negative prime patterns We analyze the negative prime patterns of the data set in the same manner. There exists no negative prime pattern including TGF_α. This is easily verified in Figure 5.2 as we can never observe a negative readout at S6K when the stimulus at TGF_α is active.

There are no negative prime patterns of degree one including IGF and only a single degree-two prime pattern that consists of the literals `pat(2,1)` and `pat(4,1)`, standing for the active stimulus at IGF and the inhibition at PI3K.

5.1.3 Discussion

Our study on perturbation experiments gives an insight on how LAD and our software package `AnswerSetLAD` can be used to analyze data sets of protein activation in signaling networks. We used prime patterns to investigate the pathway structure of the protein network of EGFR and saw that they nicely display the underlying structure of the network.

This application uses only a small part of the LAD functionalities and does not reveal any new knowledge about the network structure. However, we were able to represent the known network structure by the prime pattern analysis. We, therefore, believe that this study can be used as a basis for new ideas with respect to the investigation of protein interactions. The data analysis with LAD seems to be a promising concept in the area of signaling networks.

5.2 Synthetic biology - cell classifier circuits

The following study is published work with Hannes Klarner, Melania Nowicka and Heike Siebert [17] and based on a cooperation with Niko Beerenwinkel and Yaakov Benenson from ETH Zurich. It deals with an application of Boolean classifiers in synthetic biology making use of the ASP framework.
The study was conceived by Heike Siebert, Hannes Klarner and me. Hannes Klarner and I designed and implemented the software and started the work on the case studies and the classifier evaluation according to [78]. Melania Nowicka extended the framework with the full evaluation procedure and performed the final case studies. Hannes Klarner performed the simulated data analysis. All work done was supervised by Heike Siebert.

5.2.1 Biological background

One of the major challenges of our time is to fight diseases that cause massive damage to the human body such as cancer. Unlike only some decades ago, we now have advanced possibilities regarding medical engineering and with that we have the opportunity to actually tackle these diseases. The field of synthetic biology is rapidly growing and lots of research is done in this area [62, 42]. Cell classifier circuits are of special interest and studied intensively [99, 62]. Those synthetic devices are built in the laboratory combining certain biological parts mimicking an electronic circuit and then transferred into living cells via a plasmid or viral vector. Inside the cell the classifier circuit “decides” depending on the specific cellular markers, such as the miRNA fingerprint of the cell, whether or not it is a healthy or a diseased cell. The outcome of the classification triggers a controlled production output and leads the diseased cells to apoptosis while not harming the healthy cells. Such cell classifiers, which are still a futuristic idea, fall into the field of personalized medicine. It is a vivid and promising research area [58, 71, 78]. While the state-of-the-art methods to fight cancer, such as chemotherapy, tend to not only kill the cancerous but also a lot of healthy cells and, therefore, cause damage themselves, personalized cell classifiers would be able to remove the cancer without affecting the rest of the body. To realize this approach a first step is to take samples of a patients cancerous and healthy cells and then convert the implicitly included information into a synthetic circuit as described before. This complex task is a classical machine learning problem [102, 78]. Many of the biological building blocks of the synthetic circuits are assembled as logic gates [98, 97], which makes Boolean modeling a natural and well-suited choice.

Whereas existing approaches often only include an implicit logical formalization [104, 79, 78], we designed an approach using Boolean functions that outperforms the heuristic methods on real-life data sets regarding the solution size and provides us not only with one possible result but all globally optimal classifiers.

5.2.2 The framework - miRNA expression profiles and Boolean classifiers

This study was conducted based on discussions with the group of Niko Beerenwinkel and Yaakov Benenson from ETH Zurich. We used real-life data sets presented by Farazi et al. [44] and preprocessed by Mohammadi et al. [78]. For benchmarking and cross-validation we used self-generated input data sets. We describe the structure of the input data in the following paragraph.

The input data: miRNA expression profiles An input data set for our approach consists of rows including binarized miRNA expression profiles of different samples. miRNAs are small non-coding RNA molecules that are involved in gene regulation within a cell. The deregulation of many miRNAs has been associated with diseases such as cancer. Each row in a data set provides the `ID` of the sample, which is the labeling of the samples for identification, followed by `Annots`, which for each sample is either 0 for healthy samples or 1 for cancerous samples, and the discretized levels of measured miRNA expressions. Here 1 refers to a `high` level of the given miRNA and 0 to a `low` level. The miRNAs are named by a letter `g` followed by an integer that uniquely identifies a miRNA. An example is shown in Figure 5.3.

ID	Annots	g1	g2	g3
1.	0	1	1	0
2.	0	0	1	0
3.	1	1	0	1

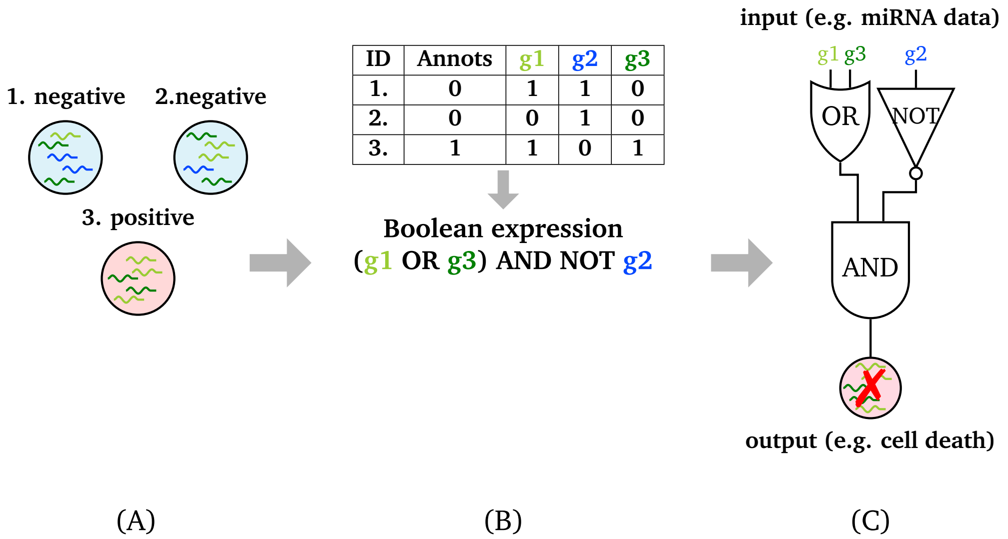
Fig. 5.3.: Example data set consisting of 2 negative and 1 positive sample.

While, of course, the discretization step of the input data plays a major role in the process, we did not work on this part of the methodology within this study.

Boolean classifiers in conjunctive normal form

As discussed before, our mathematical goal is to construct a function that is able to make a reasonable prediction for new samples based on the knowledge gained from the input data. This problem fits perfectly into the field of applications of LAD as discussed intensively in the previous chapters. However, due to the further processing, namely building the classifier circuits in the laboratory, certain constraints have to be fulfilled that do not suit the LAD approach in the standard way. Therefore, we developed a new ASP based workflow that does generate Boolean classifiers in Conjunctive Normal Form (CNF).

The structure of the synthetic cell circuits Synthetic gene circuits are designed consisting of logic gates [98, 97] within a Boolean framework. There are typically three types of gates that are used. These are disjunctions (OR), conjunctions (AND) and negations (NOT). The miRNA expression levels are used as inputs to the OR and NOT gates and then combined by an AND gate, which yields the desired output (see Figure 5.4).



ID	Annots	g1	g2	g3
1.	0	1	1	0
2.	0	0	1	0
3.	1	1	0	1

Fig. 5.4.: From miRNA expression profiles to synthetic cell circuits. (A) The inputs are miRNA expression profiles identifying the cell state as healthy or diseased. (B) The continuous miRNA levels are binarized. They can then be combined into logic gates. A Boolean function can be defined by the conjunction of logic gates that allows classifying the cell state. (C) The input signals are processed by a synthetic regulatory network. The response of the network may be a controlled production of a desired RNA or protein output leading to cell apoptosis depending on the cell state.

Different from the standard LAD approach, which we described and evaluated in the previous chapters, this demands the underlying Boolean function to be a conjunction of clauses where each clause is a disjunction of negated and non-negated inputs. Therefore, in this study we refer to a *classifier* as a Boolean function $f : \{0,1\}^n \to \{0,1\}$, where n is at most the number of considered miRNAs in a profile, in Conjunctive Normal Form (CNF). A classifier that can separate the given samples perfectly, meaning that each sample that is annotated as healthy is classified as healthy by the classifier, and each sample that is annotated as cancerous is classified as cancerous, is called a *perfect classifier*. Since we are considering real-life data sets it sometimes makes sense to allow some latitude in the accuracy due to errors that might occur in the process of gaining the input data. We make a step in this direction by taking also classifiers into account that are not perfect, but make some false positive or false negative errors. We call such classifiers *imperfect classifiers* in the following. An example for a perfect classifier of the data set shown in Figure 5.3 is:

$$(g1 \vee g3) \wedge \neg g2,$$

where $\wedge$, $\vee$ and $\neg$ represent logical conjunction, disjunction and negation, respectively. To classify a new sample as cancerous (positive) according to this classifier miRNA-g1 or miRNA-g3 should have a high expression and miRNA-g2 should have a low expression.

Classifier constraints Not only do the classifiers have to separate the given data set and fulfill the instruction of being in Conjunctive Normal Form, there are more constraints on the Boolean function that ensure that the resulting classifier can be assembled in the laboratory.

Based on the discussions with Niko Beerenwinkel and Yaakov Benenson and the work of Mohammadi et al. [78] we defined a framework of classifier constraints that force the Boolean function to a form that can be biologically engineered.

Smaller classifiers are easier to assemble, which leads to upper bounds on the overall number of inputs and the overall number of gates. According to [78] only two types of gates can be built in the laboratory. We defined those two *gate types* by the lower and upper bounds on positive and negative inputs. `Gate type 1` includes up to two positive inputs and no negative input and `gate type 2` is a single negative input. The number of occurrences of the gate types is bounded by two for type 1 and by four for type 2. The set of these *core constraints* is shown in Figure 5.5.

As mentioned before, these constraints were designed based on the specific setting in the laboratory. The constraints can easily be adjusted according to different preferences.

```
gate type 1:
        lower bound positive inputs: 0
        upper bound positive inputs: 3
        lower bound negative inputs: 0
        upper bound negative inputs: 0
        upper bound occurences:      2
gate type 2:
        lower bound positive inputs: 0
        upper bound positive inputs: 0
        lower bound negative inputs: 0
        upper bound negative inputs: 1
        upper bound occurences:      4
upper bound gates:        6
upper bound inputs: 10
```

Fig. 5.5.: A full set of core classifier constraints. A classifier that satisfies the core constraints may consist of up to 6 gates with up to 10 inputs in total. Gates are either of Type 1 (OR gate) or Type 2 (NOT gate). Gates of Type 1 may include only non-negated inputs. Gates of Type 2 may be only a single negated input.

To make the setting more flexible we added two constraints that bound the number of false positives and the number of false negatives that the classifier is allowed to make. This goes in line with the concept of *imperfect classifiers* and helps us to find solutions for problems where no perfect classifier exists or we want to allow certain types of errors in the classification due to possible errors in the data.

To ensure that each miRNA appears only once within the classifier, we added a constraint to forbid repetitions of miRNAs over all gates. This *unique-input* constraint might be relevant for increasing the robustness of the classifier against error in the miRNA expressions.

Finding optimal classifiers

A data set consisting of samples of miRNA expression profiles can be interpreted as a partially defined Boolean function (pdBf). If the underlying set of samples is contradiction-free, meaning that there is no overlap between samples that are annotated as healthy and those which are annotated as diseased, there exists at least one perfect classifier to the problem described above. In practice it is interesting to choose from the set of feasible classifiers the one that optimizes a certain cost function. This cost function can represent the actual work cost to assemble the circuit in the laboratory or for example weight specific miRNAs according to how they might interfere with other components.

We implemented four different optimization strategies that arise from the general idea of keeping the classifiers as simple as possible:

(Opt1) Minimize the number of inputs.
(Opt2) Minimize the number of gates.
(Opt3) Minimize the number of inputs followed by the number of gates.
(Opt4) Minimize the number of gates followed by the number of inputs.

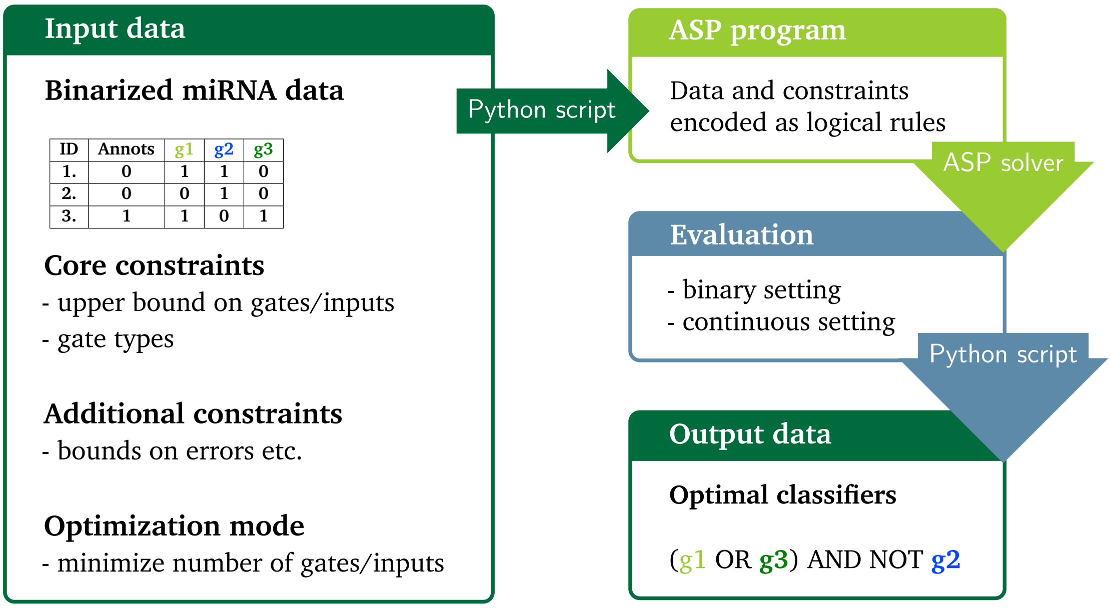
Obviously the outcome for each of the four optimization strategies is different in general. In practice it is useful to run several optimization strategies and decide on the best result based on biological expertise.

Constraint relaxation - Handling non-perfect cases As mentioned before it might not be possible to find a perfect classifier at all. For those cases we added the constraints on the number of false positives and false negatives that the classifier is allowed to make. In practice we tackle those problems by first applying the above mentioned optimization strategies to find a perfect classifier. If such a classifier does not exist, we stepwise increase the number of errors allowed until finding an imperfect classifier making the fewest errors possible. Depending on the underlying data and the application one might allow only false positives, false negatives or both types of errors. We call this procedure *constraint relaxation*.

5.2.3 The ASP encoding

As discussed in Chapter 3 the ASP environment is well-suited for constraint-based enumeration of optimal Boolean functions. Our software for the generation of Boolean classifiers in conjunctive normal form is available on GitHub [16]. It is implemented using `python` scripts. The script `classifier.py` translates the input data file into an ASP program, which can then be solved by `clingo` [49]. Additionally, it creates an image of the resulting classifier as a directed graph. A schematic overview of the workflow can be seen in Figure 5.6.

We describe the ASP encoding that is generated by `classifier.py` out of a `csv`-file including the tissue data as in Figure 5.3 in more detail in the following paragraphs.



ID	Annots	g1	g2	g3
1.	0	1	1	0
2.	0	0	1	0
3.	1	1	0	1

Fig. 5.6.: The schematic overview of our ASP-based approach to synthetic gene circuit design.

Input

Data To make the input readable for the ASP solver `clingo`, we translate the given data into facts. For each sample a predicate `tissue` is introduced having as first variable the tissue ID and as second either `healthy` or `cancer` depending on the annotation of the sample. For each sample the predicate `data` is generated. It contains three variables. The first one is again the ID number, the second one is the name of the miRNA and the third one is a Boolean variable, which is either `high`, if the miRNA expression has the binary value 1 or `low` if it has the binary value 0. The input example in Figure 5.3 translates that way to the following ASP encoding:

```
tissue(1,healthy). tissue(2,healthy). tissue(3,cancer).

data(1,g1,high). data(1,g2,high). data(1,g3,low).
data(2,g1,low). data(2,g2,low). data(2,g3,high).
data(3,g1,low). data(3,g2,high). data(3,g3,low).
```

The next lines of the code introduce the predicates `is_tissue_id` and `is_mirna` for variable binding. They allow us to iterate over all tissues and miRNAs in the rest of the program.

```
is_tissue_id(X) :- tissue(X,Y).
is_mirna(Y) :- data(X,Y,Z).
```

Constraints As explained in the previous paragraphs we implemented the option of defining constraints on the classifier in terms of bounds on the number of inputs and gates. These bounds are given to the ASP solver as facts and can be adjusted according to the specific application or preferences.

```
lower_bound_inputs(1). upper_bound_inputs(10).
lower_bound_gates(1). upper_bound_gates(2).
```

The user can define different gate types. They always include a lower and an upper bound on the positive inputs, a lower and an upper bound on the negative inputs and an upper bound on the occurrence of this gate type. Here again for variable binding the predicate `is_gate_type` assigns an index to the gate type that then uniquely connects the bounds to it.

```
is_gate_type(1).
lower_bound_pos_inputs(1, 0). upper_bound_pos_inputs(1, 2).
lower_bound_neg_inputs(1, 0). upper_bound_neg_inputs(1, 0).
upper_bound_gate_occurrence(1, 1).
```

Finding a feasible classifier

Choosing gates The first step on the way to a feasible classifier is to choose a number of gates in the range of the lower and upper bound on the gate number.

```
1 {number_of_gates(X..Y)} 1 :- lower_bound_gates(X),
upper_bound_gates(Y).
is_gate_id(1..X) :- number_of_gates(X).
```

The predicate `number_of_gates` exists exactly once and includes one variable whose value is a number between the lower and the upper bound on the gates. The predicate `is_gate_id` is used for variable binding.

To each of the gates we assign exactly one gate type out of the gate types defined in the input section.

```
1 {gate_type(GateID, X): is_gate_type(X)} 1
:- is_gate_id(GateID).
```

Next, gate inputs are chosen from the set of miRNAs. The predicate `gate_input` contains the variables. The first is a number that represents the gate ID, the second is a Boolean variable that is either `positive` or `negative` depending on whether we choose the non-negated or negated input of the miRNA and the third one is the name of the miRNA itself. The two possible values for the sign are accessible via the predicate is_sign.

```
X {gate_input(GateID, positive, MiRNA): is_mirna(MiRNA)} Y
:- gate_type(GateID, GateType),
lower_bound_pos_inputs(GateType, X),
upper_bound_pos_inputs(GateType, Y).
is_sign(positive). is_sign(negative).
```

At this point of the encoding it is possible for a miRNA to appear both as positive and negative input to the same gate, which would lead to a gate that is active in all cases. This is not useful in both the mathematical and biological sense. Therefore, we added the following *one-sign-only constraint* that allows each miRNA to appear only positive or negative in one gate.

```
{gate_input(GateID, Sign, MiRNA): is_sign(Sign)} 1
:- is_mirna(MiRNA), is_gate_id(GateID).
```

If we are interested in classifiers that contain each miRNA at most once, for reasons of robustness as explained before, for example, we might drop this constraint and use the *unique-input constraint* only, which we describe below.

The last clause on the gates takes the bound on the total number of occurrences of each gate type into account. Each gate type can appear at most X times, where X is the upper bound on the gate occurrence.

```
{gate_type(GateID,GateType): is_gate_id(GateID)} X
:- upper_bound_gate_occurence(GateType,X).
```

Choosing miRNA inputs Within the following lines of the code miRNA inputs are assigned to the gates. The first constraint ensures that each chosen gate gets assigned at least one input.

```
1 {gate_input(GateID,Sign,MiRNA):
is_sign(Sign), is_mirna(MiRNA)}
:- is_gate_id(GateID).
```

This constraint is important because a lower bound of 0 on the positive or the negative inputs might lead the solver to create a solution including empty gates, which is meaningless in the mathematical and biological sense.

The next line in the encoding is the *unique-input constraint*, which might be dropped in exchange for the *one-sign-only constraint* as described before. The *unique-input constraint* allows each miRNA to appear only once over the whole classifier.

```
{gate_input(GateID,Sign,MiRNA):
is_sign(Sign), is_gate_id(GateID)} 1
:- is_mirna(MiRNA).
```

The number of total inputs used has to lie within the defined range of the bounds `lower_bound_inputs` and `upper_bound_inputs`. Therefore, we add another cardinality constraint.

```
X {gate_input(GateID,Sign,MiRNA):
is_gate_id(GateID), is_sign(Sign), is_mirna(MiRNA)} Y
:- lower_bound_inputs(X), upper_bound_inputs(Y).
```

Firing of gates and prediction of the classifier The following lines of the code are used to evaluate whether a gate fires for one of the given samples or not. The predicate `gate_fires` for a gate ID and a tissue ID exists, if the expression level of the miRNA is in line with the input for the gate. This is the case if the miRNA expression level is `high` and the input is `positive` or if the expression level is `low` and the input is `negative`. Note here again that we are searching for a classifier in Conjunctive Normal Form. Each gate is, therefore, a disjunction of its inputs and it is sufficient for one of the inputs to be satisfied on the data to make the gate fire.

```
gate_fires(GateID,TissueID)
:- gate_input(GateID,positive,MiRNA),
data(TissueID,MiRNA,high).

gate_fires(GateID,TissueID)
:- gate_input(GateID,negative,MiRNA),
data(TissueID,MiRNA,low).
```

The full classifier is then a conjunction of its gates. This means that all its gates have to fire to make the classifier predict `cancer`. If any of the gates in the classifier do not fire for a sample, the classifier will predict the tissue to be `healthy`.

```
classifier(TissueID,healthy)
:- not gate_fires(GateID, TissueID), is_gate_id(GateID),
is_tissue_id(TissueID).

classifier(TissueID,cancer)
:- not classifier(TissueID, healthy), is_tissue_id(TissueID).
```

Because the outcome of the classifier is binary, i.e. either `cancer` or `healthy`, we can construct the predicate `classifier(TissueID,cancer)` by negating the previously introduced predicate `classifier(TissueID,healthy)`.

Consistency of classifier and data The last step of the code is important to make the classifier consistent with the data. The two following integrity constraints forbid answer sets including a classifier that predicts `cancer` for a `healthy` tissue and the other way around. That way perfect classifiers for the input data set are generated.

```
:- tissue(TissueID,healthy), classifier(TissueID,cancer).
:- tissue(TissueID,cancer),  classifier(TissueID,healthy).
```

5.2.4 Results

We tested our method on five data sets consisting of tissue data of breast cancer cells that were presented by Farazi et al. [44] and discretized and examined by Mohammadi et al. [78]. This way we were able to compare our results directly to the later study. This part of the work was started by Hannes Klarner and me and then continued, completed and edited by Melania Nowicka. Here we present a summary of the results.

Case studies

The five data sets all contain samples of breast cancer cells of patients, which are labeled as *positive* (cancerous) or *negative* (healthy). Each sample consists of the binarized expression levels of around 400 to 500 miRNAs. The discretization is the result of the application of a data set dependent threshold. All miRNA expression levels above this threshold were assumed to be high (1) and all expression levels below this threshold were assumed to be low (0). A summary of the details on the five breast cancer data sets is given in Table 5.2.

Subtype	Samples	Positive	Negative	miRNA	BinThreshold
All	178	167	11	478	250
Triple-	82	71	11	456	250
Her2+	86	75	11	438	1250
ER+ Her-	32	21	11	392	1250
Cell Line	17	6	11	375	50

Tab. 5.2.: The Breast cancer datasets. The first column shows the overall number of samples, the second column the number of positive and the third column the number of negative samples. Column four includes the number of miRNAs taken into account and the last column shows the binarization threshold applied for data binarization.

The *Breast Cancer All* data set is a combination of the other four data sets. Repeated samples were deleted. For all of the five data sets we applied the same procedure of searching for a perfect classifier according to the *core constraints* (see Figure 5.5) minimizing the number of inputs followed by the number of gates (Opt 3). Only one data set could be separated by a perfect classifier, namely the data set *Breast Cancer Cell Line*. In all the other four cases we applied the *constraint relaxation* procedure. We describe the results in more detail in the following paragraphs.

Breast Cancer All In Figure 5.7 two classifiers for the combined data set *Breast Cancer All* are shown. The one on the left hand side is an optimal classifier consisting of only one gate of type 2 with *miR-378* as negated input. The classification leads to four false negative and three false positive errors on the data set. While this classifier is as short as possible and, therefore, easy to assemble in the laboratory, it is worth considering larger classifiers due to cancer cell diversity.

One way of approaching this problem could be to forbid a type of error as mentioned before. We exemplarily show this workflow here by not allowing any false positives to occur. This results in six optimal classifiers, which are all isomorphic to the one presented in Figure 5.7 on the right hand side. Note that while here it was possible to find the isomorphism class by hand this problem might become very large for certain data sets. A post processing step might be needed for those situations.

Note that in both classifiers shown *miR-378* appears as a negated input to a gate of type 2. In [44] this miRNA was identified to be low expressed and reasonable as marker for classification. An unrelated study [84] stated that e.g. *miR-144* is down-regulated in breast cancer. This miRNA appears in the classifier on the right

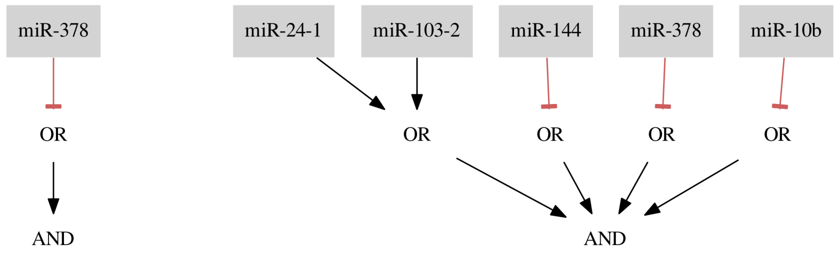


Fig. 5.7.: Classifiers for the Breast Cancer All data set.

hand side as negated input. Both of the classifiers seem to be useful and it should be decided depending on the situation at hand which of the two is chosen.

Breast Cancer Triple- No perfect classifier exists for the *Breast Cancer Triple-* data set. The *constraint relaxation* procedure provides us with two imperfect classifiers allowing three false negatives and two false positives. These are the fewest errors possible. The two classifiers are shown in Figure 5.8. Both of the classifiers consist of two inputs in two different gates. They both contain *miR-378* as negated input in a gate of type 2. The first classifier has another gate of type 2 with the input *miR-144*. The second classifier has a second gate of type 1 with *miR-24-1* as non-negated input. Both classifiers seem to be reasonable as *miR-378* was recognized to be down-regulated in cancer by [44] and *miR-24-1* and *miR-144* were found to be up-regulated by [91] and [84], respectively.

To decide for one of the two classifiers in practice, a natural approach would be to check the reliability of the binarized miRNA measurements for the differing miRNAs and to weigh the cost of building both classifiers in the laboratory.

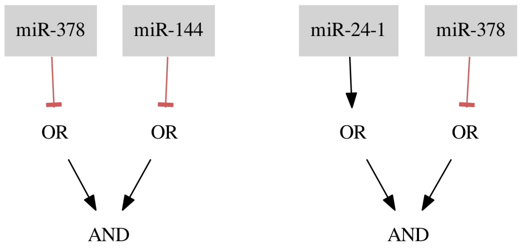


Fig. 5.8.: Classifiers for the breast cancer *Triple-* data set.

Breast Cancer Her2+ The data set *Breast Cancer Her2+* could not be perfectly separated. Therefore, we applied *constraint relaxation*. It was sufficient to allow the classifier to make only one mistake, namely a false positive error, to find a solution. The resulting classifier is shown in Figure 5.9.

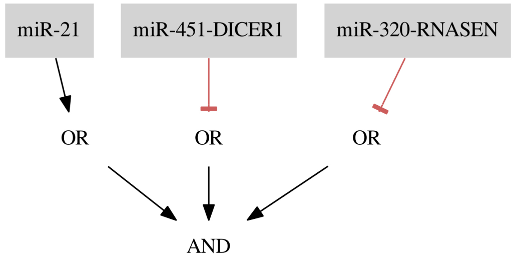


Fig. 5.9.: Classifier for the breast cancer *Her2+* data separating samples with one false positive error.

It consists of two gates of type 2, one of them having the *miR-451-DICER1* and one of them having *miR-320-RNASEN* as negated inputs. The third gate is a gate of type 1 with *miR-21* as single non-negated input. In [44] the authors marked *miR-451-DICER1* and *miR-320-RNASEN* as down-regulated and *miR-21* as up-regulated in *Her2+* breast cancer. Our result fits this claim.

Breast Cancer ER+ Her- For the data set *Breast Cancer ER+ Her-* we present two results in Figure 5.10. The classifier on the left hand side is the optimal classifier consisting of two gates with one input each. Its classification results in two false positive errors. The classifier shown on the right hand side is shorter having only one gate with one input. This classifier makes three false positive errors on the data set. Nevertheless, in practice it is reasonable to take this result and the approach of further relaxing the constraints into account as it may result in shorter and easier to assemble classifiers. In that case one should take a closer look on the data to see if the extra error is a reliable sample or not.

Both classifiers shown contain *miR-21*, which was marked up-regulated by [44]. The optimal classifier additionally includes *miR-320-RNASEN* as negated input to a gate of type 2, which was marked down-regulated in the same study.

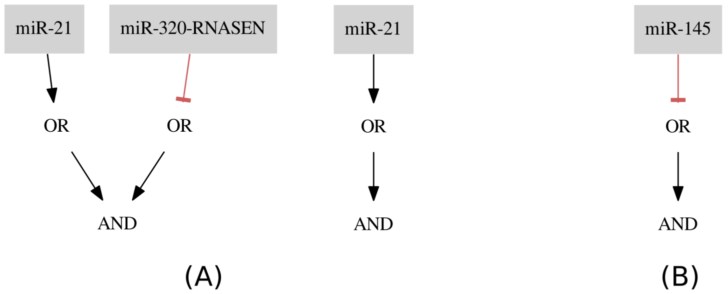


Fig. 5.10.: Results for the breast cancer *ER+ Her-* (A) and *Cell Line* (B) data sets.

Breast Cancer Cell Line The data set *Breast Cancer Cell Line* was the only one of the five data sets that is perfectly separable. We found six perfect classifiers for this data set. All of them consist of one gate with one input only. Five of the perfect classifiers contain a gate of type 2, having a miRNA as negative input, and one perfect classifier has a type 1 gate with one miRNA as positive input. All classifiers are listed in Table 5.3.

We present one of the perfect classifiers as an example in Figure 5.10. This classifier uses the miRNA *miR-145* as a single negated input. This means that each sample including *miR-145* with a low expression level will be classified as cancerous. This is in line with a study by Farazi et al. [44] where *miR-145* is marked as down-regulated in cancer cells.

Subtype	Classifier	FN rate	FP rate	AUC	Ma	Mw
BC All	(¬ miR-378)	0.02	0.27	0.96	0.79	-0.47
Triple-	(¬ miR-378) ∧ (¬ miR-144)	0.04	0.18	0.98	0.81	-0.34
	(miR-24-1) ∧ (¬ miR-378)	0.04	0.18	0.99	0.61	-0.10
Her2+	(miR-21) ∧ (¬ miR-451-DICER1) ∧ (¬ miR-320-RNASEN)	0.00	0.09	0.99	0.87	-0.24
ER+ Her-	(miR-21)	0.00	0.27	1.00	0.85	0.14
	(miR-21) ∧ (¬ miR-320-RNASEN)	0.00	0.18	0.96	0.74	-0.46
Cell Line	(¬ miR-145)	0.00	0.00	1.00	1.66	1.33
Cell Line	(¬ miR-143)	0.00	0.00	1.00	1.66	1.33
Cell Line	(¬ miR-199a-2-5p)	0.00	0.00	1.00	1.66	1.33
Cell Line	(¬ miR-451-DICER1)	0.00	0.00	1.00	1.66	1.33
Cell Line	(¬ miR-146a)	0.00	0.00	1.00	1.66	1.33
Cell Line	(¬ miR-425)	0.00	0.00	1.00	1.66	1.33

Tab. 5.3.: Evaluation of our breast cancer classifiers. We show the false negative rate, false positive rate, AUC, average margin and worst margin.

5.2.5 Performance analysis and testing

Classifier evaluation

In [78] the authors developed a scoring for classifiers on data sets of the given kind. The scoring is based on a mathematical interpretation of a synthetic regulatory network and allows one to estimate the circuit output concentration for the continuous miRNA data and a Boolean classifier. This scoring is used to estimate whether the concentration of the output will be sufficient to cause, for example, cell death.

We followed their approach to compare our results and implemented a python script called `scores.py` to evaluate our classifiers. Within the script we calculated the false negative rate (FN rate), false positive rate (FP rate), the area under the curve (AUC), average margin (Ma) and worst margin (Mw). For the calculations we kept the same biochemical parameter sets and binarization thresholds as proposed by [78]. While the authors only considered the last three scores we added the false positive and false negative rate to compare results in the binary setting. For the comparison one should keep in mind that Mohammadi et al. [78] used a different modeling framework with the discretization error as one of their optimality criteria while our classifiers are only optimized on the binary data sets. The central goal of both approaches to find a minimal classifier is the same, nevertheless. In Table 5.3 our results are shown.

In all cases we were able to find shorter classifiers and in most cases improve the accuracy of classification in the binary setting. Otherwise, the accuracy is identical. In the continuous setting we obtained comparable results. However, interpretation of these scores linking the Boolean to the continuous classifier are difficult to assess. For a detailed comparison with the results of [78] see our publication [17].

Benchmarking

To test our approach we evaluated our method on simulated data sets. After a collective discussion, this part of the study was mostly done by Hannes Klarner. Because the results substantiate the usefulness and show the scalability of our software, we here present a summary of the main results.

All computations, for both the benchmarking and the cross-validation, were performed on a Linux AMD64 with 2.83GHz and 32GB of memory.

Data generation Random 0-1 matrices were generated with each entry independent and 0 and 1 equally likely to occur. We chose the dimension of the matrices starting from 10×10 to 500×500 with step size 10 both for the rows and the columns. The benchmark, therefore, consists of $50 \times 50 = 2500$ data points.

For each matrix we generated an annotation classifier to decide which of the samples is labeled positive and which negative. We used two different setups for the generation of this *annotation classifier*. The first setup guarantees the existence of a solution by choosing the annotation classifier such that it satisfies the *core constraints* (see Figure 5.5) by choosing with equal probability each gate and input.

The second setup constructs an annotation classifier uncoupled from any constraints by using binomial distribution. We defined the number of gates to be $\lfloor n/10 \rfloor$ and the maximal number of inputs per gate as 5. Each gate and input is then chosen with equal probability. The number of gates and inputs is arbitrary. The setup was chosen to give an insight on how the algorithm performs when a solution is not guaranteed.

Experiments In Figure 5.11 the resulting heat maps for Optimization strategy 3 (Opt 3) for both the first feasible and the optimal solution for Setup 1 and 2 are shown. We used time-outs between 10 minutes to 1 hour.

Figure 5.11 (A) shows the time needed to find a feasible solution for Setup 1. This graphic illustrates that the problem of finding a feasible classifier in the scenario where the existence of a classifier satisfying the core constraints is guaranteed, is easy to solve for our ASP program. Figure 5.11 (B) visualizes the time needed to compute an optimal solution for the same setup. About 16% of the problems were timed out. We see here that the number of time-outs increases with the number of miRNAs but does not seem to depend on the number of tissues. We believe that this is due to the exponentially growing search space with the number of miRNAs while additional samples might only add redundant information after a certain size is reached.

The heat maps for Setup 2 are shown in Figure 5.11 (C,D). The squares outlined in black indicate that it was proven within the time limit that the problem is infeasible. Nearly all the problems shown in Figure 5.11 (C) could be solved within the given time limit. We see that the probability that a feasible solution exists grows with the number of miRNAs. This behavior could be due to the fact that each miRNA that is added to the search space might be the one that is needed as an input for the feasible classifier. The expression levels of the miRNAs are chosen to be independent from each other. However, in real life data we would not expect this behavior as miRNAs and their expression levels are likely to be dependent on each other. The heat map in Figure 5.11 (D) displays the same results with the difference that optimal solutions are harder to calculate and, therefore, the time-out was reached for nearly all of the larger data sets.

Cross-validation

Additionally to the benchmarking we performed a 10-fold cross-validation for Setup 1, which guarantees the existence of a feasible classifier satisfying the core constraints. The results are shown in Figure 5.12.

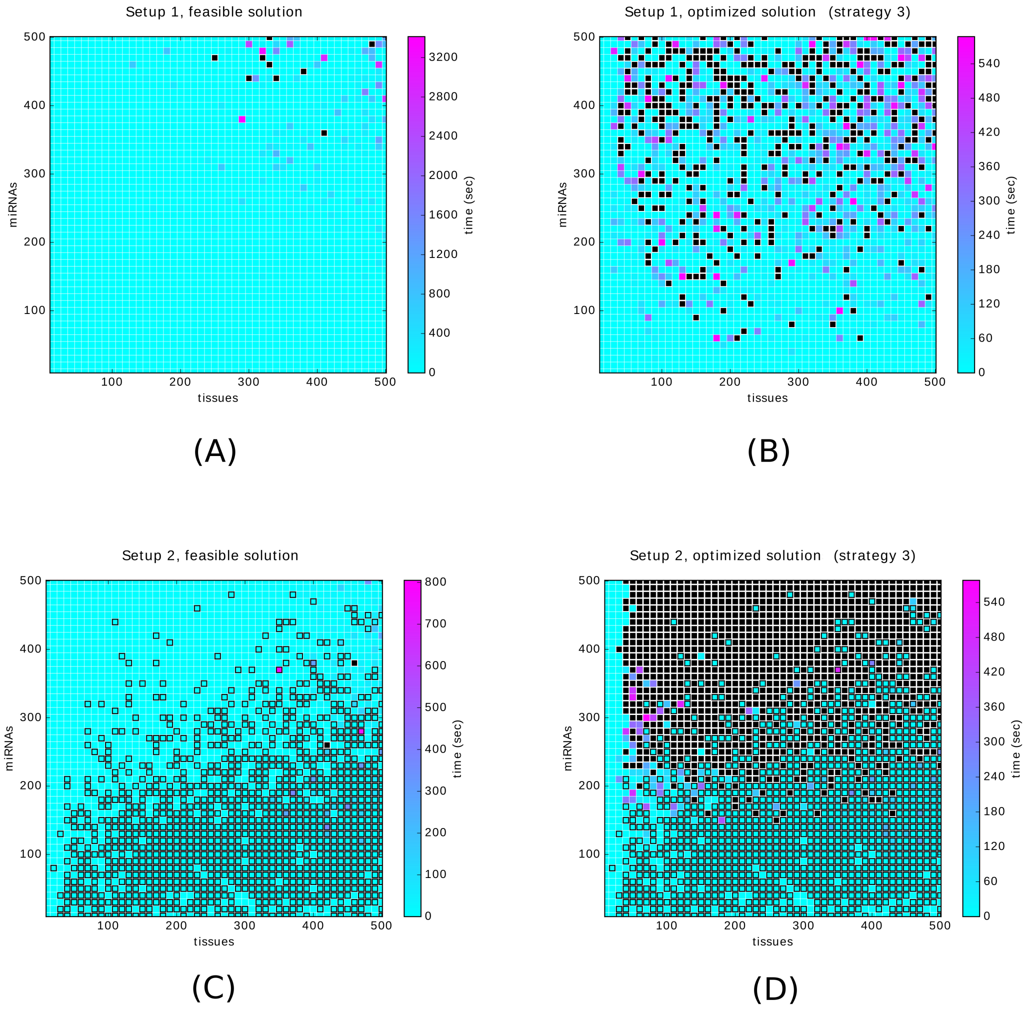


Fig. 5.11.: The results of the benchmarking. Squares filled in black indicate that the time-out was reached. Squares outlined in black indicate that the infeasibility of the problem was proven within the time limit. Figure (A) and (B) are results of Setup 1, where the existence of a classifier satisfying the core constraints is guaranteed. Figure (A) shows the time to compute a feasible and Figure (B) the time to compute an optimized solution, respectively. Figure (C) and (D) are results of Setup 2, where the existence of a classifier satisfying the core constraints is not guaranteed. Figure (C) shows the time to compute a feasible and Figure (D) the time to compute an optimized solution, respectively. Optimization strategy 3 (first gates, then inputs) was used in both setups.

Each of the data sets represented by a point in the heat map was divided in ten parts of equal size. A classifier was then built depending on nine out of the ten parts and tested on the remaining one. We did this ten times for each of the ten parts of the data set and added up the resulting running times. This sum of the running times divided by ten is shown in the heat maps in Figure 5.12. Time-outs were treated as false predictions.

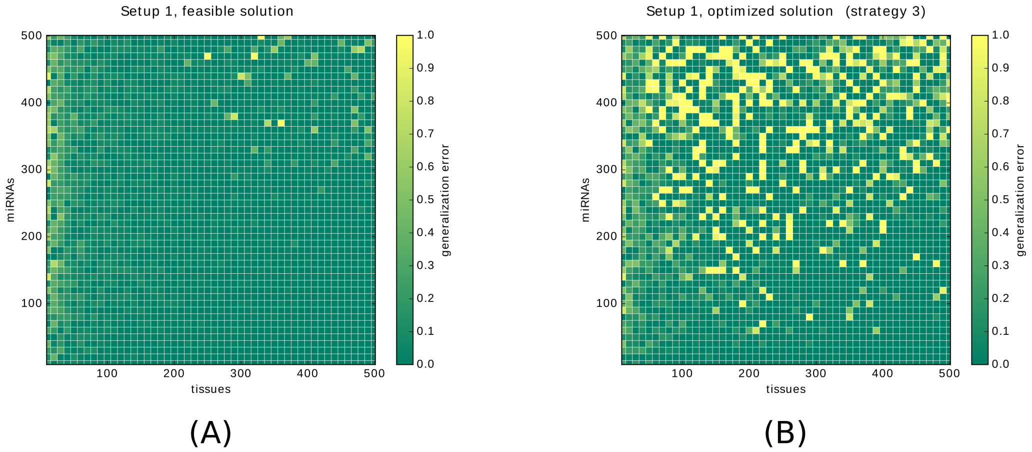


Fig. 5.12.: The results of the cross-validation for Setup 1. Figure (A) shows the cross-validation for finding a feasible solution. Figure (B) shows the time for finding an optimized solution with strategy 3 (first gates, then inputs). Time-outs were treated as false predictions.

In Figure 5.12 (A) two areas of increased error rates can be seen: a vertical strip on the left and the circular area in the top right corner. We think that the reason for the first one is that a certain number of samples is needed to make a classifier reliable. In our case these are around 30 samples. Otherwise the missing examples of positive and negative observations leave too much room for false classifications. The circular error region in the top right corner is due to the time-outs that were counted as misclassification. The bigger the problem the likelier it is to be to difficult to be solved in time. Our hypothesis is that given unlimited time the error rate will tend towards zero because the existence of a solution is guaranteed from the setup.

Figure 5.12 (B) shows the same behavior of the program except for the fact that optimal solutions are more difficult to be calculated and, therefore, a lot more problems are timed-out when the number of miRNAs increases.

5.2.6 Discussion

This study shows the power of Boolean functions combined with the ASP environment for a pivotal application from synthetic biology.

We implemented a workflow for classifier optimization to obtain globally optimal perfect and imperfect (in the case when no perfect classifier exists) classifiers. The constraints used for the classifier design are real-life requirements for making the fabrication of the circuit possible in the laboratory.

We were able to show that our approach outperforms the heuristic method from Mohammadi et al. [78] regarding the size of the computed classifiers in five real-life data sets. Furthermore, we achieved comparable scores for our classifiers according to a scoring scheme developed in [78] although we did not optimize our classifiers according to the same criteria.

In the first case study we mentioned the problem of the same solution appearing several times in permuted form. These results are due to the setup of our program. The encoding shows that, for example, IDs are assigned to gates for binding. Any permutation of these IDs will result in a different solution for the ASP solver. Anyhow, all of these solutions lie in the same isomorphism class. Breaking such symmetries within an ASP program is a topic on its own and we did not investigate solving it for our approach as it was still easily possible to find the isomorphism classes by hand. Nevertheless, it would be useful to tackle this problem or add a postprocessing step to the approach to facilitate the method.

For situations where no perfect classifier exists, we explained our procedure of *constraint relaxation*. In perspective work, it might be of interest to use `asprin` [29] to search for Pareto-optimal solutions as discussed in Chapter 3.

For future work it might be of interest to take a closer look into the discretization step and compare the usefulness of available discretization methods [46]. The data used here was preprocessed by Mohammadi et al. [78]. It could be useful to weigh the miRNAs according to the reliability of their binarization. This way more reasonable miRNAs could be chosen for the classifier. The five data sets are imbalanced regarding positive and negative samples. It is worth to investigate whether this imbalance affects the results.

The benchmark shows that our approach is capable of computing solutions for large data sets with hundreds of samples and miRNAs in the case that a solution exists. The time for the computation generally lies within a time scale of minutes on a personal computer. We, therefore, think that our method is very useful and well-suited for medical applications.

Our approach is very flexible in terms of changing the classifier constraints according to different requirements or allowing different kinds of errors depending on the problem at hand. This gives the opportunity of further developments in the field of computational classifier design in close cooperation with experimental experts.

Part V

Conclusion

Discussion | 6

The work of this thesis is located at the intersection of mathematics and computer science with life sciences. In the recent years, technological progress has improved recording and observation methods substantially, which caused a massive increase of data set size in this research field. Therefore, a major challenge is to extract the relevant information out of the untransparent mass of data. Besides developing efficient analytical methods for this purpose, it is crucial to design the analysis such that the results are understandable and insights can be shared comprehensively across the interdisciplinary field. Many of the data analysis problems in biology and medicine can be summarized by the search for attributes or attribute combinations that allow assigning an observation to a certain property. In this thesis, we focused on binary problems only.

The core idea of our work is to identify and optimize a suitable machine learning approach for data analysis in the biomedical field. Many machine learning approaches have been evaluated and used in life sciences over the last decades [45, 43, 68]. An approach, which is particularly meaningful regarding biological and medical applications [56, 3, 8, 88], is the *Logical Analysis of Data* (LAD) [35]. LAD is a powerful and well-performing machine learning method [24, 7, 54, 93] combining ideas from Boolean functions, combinatorics and optimization. We introduced the concept of LAD in detail in this thesis. Furthermore, we presented our contributions to LAD in practice, theory and application.

To allow the easy and efficient application of LAD, we developed a software package including all LAD functionalities, called `AnswerSetLAD`, which is freely available on GitHub [14]. `AnswerSetLAD` makes use of *Answer Set Programming* (ASP) [73], a declarative programming paradigm oriented towards difficult search problems and combinatorial optimization problems. This combination of LAD with ASP is an innovative idea and a major achievement of this work. We published introductory results here [15]. The framework of ASP is perfectly suitable for the search for patterns of various types in large data sets, which play a key role in LAD. Besides its efficiency, the design of `AnswerSetLAD` using ASP, provides the user with short, clear and succinct programs. This adds comprehensibility and communicability. We think that ASP is in particular meaningful for the implementation of LAD functionalities, because it preserves the original idea of LAD, which is to make information accessible across disciplines.

Our software AnswerSetLAD includes all steps of the LAD process, namely *data binarization*, *pattern generation* and *theory formation*. We compared our ASP to a state-of-the-art MILP approach for the generation of maximal patterns, which are one of the most common pattern types associated with the LAD methodology. Here we could clearly show that ASP is superior to MILP for the given task. This result corroborates our hypothesis, that ASP is a sophisticated choice as a framework for the LAD method.

Regarding the theoretical ideas of LAD, we presented two accomplishments in the field of prime pattern generation and the formation of prime theories. The identification of prime patterns is central in LAD research because of their succinct characteristics, which makes them easy to interpret. The enumeration of all prime patterns in a data set, however, is a difficult problem in general. We presented an algorithm, which allows calculating prime patterns efficiently in the case that the data set has small maximal Hamming distance between positive and negative observations. With regards to biomedical application, this property is likely to apply as the players within biological systems are not expected to act independently from each other.

The second idea regarding the theoretical advancement of LAD is work concerning the field of prime theories. Since the number of theories for a data set is large in general, it is useful to develop reasonable measures, which allow ranking the set of theories. We here proposed an approach using a statistical measure for this purpose. This measure is based on the occurrences of the different literals over the whole set of prime patterns.

In the last chapter of this thesis, we applied our work on classification with Boolean functions using ASP to two biomedical problems. The first application concerns protein interactions in signaling networks. We think that our new approach of identifying the protein structure in a cell using LAD is a promising idea and an interesting application for LAD pattern generation. In the case of the EGFR signaling network we successfully revealed patterns that reflect the structure of the system. Nevertheless, we discussed only a small example with a lot of prior knowledge. We think it is worth putting more work in this field and expand the underlying theory.

The second application is located in the highly topical field of synthetic biology. We presented our published study on real-life breast cancer data sets [17]. Within our study we improved former heuristic methods on this topic [78] by effectively applying the ASP framework, which allows us to find globally optimal classifiers. The discussed applications show that the here presented methodology of LAD, in particular in combination with ASP, is meaningful and carries great future potential in biomedical research.

Future directions Our software package `AnswerSetLAD` provides various LAD functionalities. However, it could be fruitful to extend the software, especially regarding the theory formation step. We presented an iterative algorithm [35] to find a pattern cover for a data set. This algorithm is implemented using `python` code. We think that it is worth making use of the advantages of ASP in the theory formation step as well. As we have seen in the field of pattern generation, the resulting implementation might outperform classical approaches in runtime efficiency, especially when processing large data sets.

In Chapter 4 we discussed two ideas and implementations for theory formation in ASP. Both approaches require a full set of prime patterns for a data set for selection. This is computationally very expensive and especially for large data sets not feasible. Regarding future work it is worth putting more effort in an ASP program that finds a full pattern cover of a specific type directly without the knowledge of the whole set of patterns for the data set.

Since the number of (prime) theories for a data set is large in general it is an important challenge to narrow down the pool of solutions to a manageable size. Therefore, we believe that research in this direction is valuable. The statistical measure we defined to constrain the set of prime theories to the set of core theories is a first idea in this direction. This topic should be further investigated by testing the measure on representative data sets. It could be of interest to vary the composition of literal weights within the statistical measure or extend it to make use of subpatterns instead of literals only.

Besides future developments regarding practice and theory a promising field for further work lies in the application of LAD. `AnswerSetLAD` is an open-source software that enables the user to take all advantages of LAD and apply them to any application. As said before, the fields of biology and medicine, besides others, provide many suitable topics. We described our approach for identifying protein interactions in a signaling network. This field of application is highly interesting and offers space for future developments.

Our work on LAD in the context of biomedical research has been very successful. The combination of LAD and ASP led to a software package, which shows excellent performance and and can be easily extended with further functionalities. We saw how the application of LAD and ASP to biological data can be used for a deeper understanding of biological systems. We conclude that discrete mathematics and especially the here presented methodology of LAD, as well as its combination with ASP, are able to open new ways in biomedical research and should, therefore, be considered in further work.

Bibliography

[1] Gabriela Alexe, Sorin Alexe, David E. Axelrod, Peter L. Hammer, and Dinah Weissmann. “Logical analysis of diffuse large B-cell lymphomas”. In: *Artificial intelligence in medicine* 34 (2005), pp. 235–67.

[2] Gabriela Alexe, Sorin Alexe, Tibérius O. Bonates, and Alexander Kogan. “Logical analysis of data – the vision of Peter L. Hammer”. In: *Annals of Mathematics and Artificial Intelligence* 49.1 (2007), pp. 265–312.

[3] Gabriela Alexe, Sorin Alexe, David E. Axelrod, Tiberius Bonates, Irina I. Lozina, Michael Reiss, and Peter L. Hammer. “Breast cancer prognosis by combinatorial analysis of gene expression data”. In: *Breast cancer research : BCR* 8 (2006), R41.

[4] Gabriela Alexe, Sorin Alexe, and Peter L. Hammer. “Pattern-based Clustering and Attribute Analysis”. In: *Soft Comput.* 10.5 (2006), pp. 442–452.

[5] Gabriela Alexe, Sorin Alexe, Peter L. Hammer, and Alexander Kogan. “Comprehensive vs. comprehensible classifiers in logical analysis of data”. In: *Discrete Applied Mathematics* 156.6 (2008). Discrete Mathematics and Data Mining II, pp. 870 –882.

[6] Gabriela Alexe, Sorin Alexe, Lance A. Liotta, Emanuel Petricoin, Michael Reiss, and Peter L. Hammer. “Ovarian cancer detection by logical analysis of proteomic data.” In: *Proteomics* 4 3 (2004), pp. 766–83.

[7] Gabriela Alexe and Peter L. Hammer. “Spanned patterns for the logical analysis of data”. In: *Discrete Applied Mathematics* 154.7 (2006), pp. 1039 –1049.

[8] Sorin Alexe, Eugene Blackstone, Peter L. Hammer, Hemant Ishwaran, Michael Lauer, and Claire Pothier. “Coronary Risk Prediction by Logical Analysis of Data”. In: *Annals of Operations Research* 119 (2002).

[9] Sorin Alexe and Peter L. Hammer. “Accelerated algorithm for pattern detection in logical analysis of data”. In: *Discrete Applied Mathematics* 154.7 (2006), pp. 1050 –1063.

[10] Uri Alon. *An Introduction to Systems Biology: Design Principles of Biological Circuits*. Chapman & Hall/CRC Mathematical and Computational Biology, 2007.

[11] Naomi S. Altman. “An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression”. In: *The American Statistician* 46.3 (1992), pp. 175–185.

[12] Martin Anthony and Joel Ratsaby. “Robust cutpoints in the logical analysis of numerical data”. In: *Discrete Applied Mathematics* 160.4 (2012), pp. 355 –364.

[13] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[14] Katinka Becker. *`AnswerSetLAD` - an ASP software package for LAD*. `https://github.com/katinkab/AnswerSetLAD`. 2020.

[15] Katinka Becker, Martin Gebser, Torsten Schaub, and Alexander Bockmayr. “Answer Set Programming for Logical Analysis of Data”. In: *WCB@CP*. 2016, pp. 15–26.

[16] Katinka Becker and Hannes Klarner. *RnaCancerClassifier*. `https://github.com/hklarner/RnaCancerClassifier`. 2019.

[17] Katinka Becker, Hannes Klarner, Melania Nowicka, and Heike Siebert. “Designing miRNA-Based Synthetic Cell Classifier Circuits Using Answer Set Programming”. In: *Frontiers in Bioengineering and Biotechnology* 6 (2018), p. 70.

[18] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. *DLV - User Manual*. `http://www.dlvsystem.com/html/DLV_User_Manual.html`.

[19] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[20] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. “Occam’s Razor”. In: *Information Processing Letters* 24.6 (1987), pp. 377 –380.

[21] Tibérius O. Bonates and Vaux S. D. Gomes. *LAD-WEKA tutorial*. `https://lia.ufc.br/~tiberius/lad/`. 2014.

[22] Tibérius O. Bonates, Peter L. Hammer, and Alexander Kogan. “Maximum patterns in datasets”. In: *Discrete Applied Mathematics* 156.6 (2008), pp. 846 –861.

[23] Endre Boros, Yves Crama, Peter Hammer, Toshihide Ibaraki, Alexander Kogan, and Kazuhisa Makino. “Logical analysis of data: classification with justification”. In: *Annals of Operations Research* 188 (2011), pp. 33–61.

[24] Endre Boros, Peter L. Hammer, Toshihide Ibaraki, Alexander Kogan, Eddy Mayoraz, and Ilya Muchnik. “An Implementation of Logical Analysis of Data”. In: *Knowledge and Data Engineering, IEEE Transactions on* 12 (2000), pp. 292 –306.

[25] Endre Boros, Toshihide Ibaraki, and Kazuhisa Makino. *Boolean analysis of incomplete examples*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 440–451.

[26] Endre Boros, Toshihide Ibaraki, and Kazuhisa Makino. “Error-Free and Best-Fit Extensions of Partially Defined Boolean Functions”. In: *Information and Computation* 140.2 (1998), pp. 254 –283.

[27] Endre Boros, Toshihide Ibaraki, and Kazuhisa Makino. “Logical analysis of binary data with missing bits”. In: *Artificial Intelligence* 107.2 (1999), pp. 219 –263.

[28] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and regression trees*. Belmont, CA: Wadsworth International Group, 1984.

[29] Gerhard Brewka, James Delgrande, Javier Romero, and Torsten Schaub. “asprin: Customizing Answer Set Preferences without a Headache”. In: *AAAI* (2015).

[30] Renato Bruni. “Reformulation of the support set selection problem in the logical analysis of data”. In: *Annals of Operations Research* 150.1 (2007), pp. 79–92.

[31] Renato Bruni, Gianpiero Bianchi, Cosimo Dolente, and Claudio Leporelli. “Logical Analysis of Data as a Tool for the Analysis of Probabilistic Discrete Choice Behavior”. In: *Computers and Operations Research* (2018).

[32] Christopher Burges. “A Tutorial on Support Vector Machines for Pattern Recognition”. In: *Data Mining and Knowledge Discovery* 2 (1998), pp. 121–167.

[33] Potsdam Answer Set Solving Collection. *Potassco*. `https://potassco.org/`.

[34] Yves Crama and Peter L. Hammer. *Boolean Functions - Theory, Algorithms, and Applications*. 2011.

[35] Yves Crama, Peter L. Hammer, and Toshihide Ibaraki. “Cause-effect relationships and partially defined Boolean functions”. In: *Annals of Operations Research* 16.1 (1988), pp. 299–325.

[36] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. “Encoding Planning Problems in Nonmonotonic Logic Programs”. In: *Proceedings of the Fourth European Conference on Planning* (1997).

[37] Mathurin Dorel, Bertram Klinger, Torsten Gross, Anja Sieber, Anirudh Prahallad, Evert Bosdriesz, Lodewyk F. A. Wessels, and Nils Blüthgen. “Modelling signalling networks from perturbation data”. In: *Bioinformatics* 34 23 (2018), pp. 4079–4086.

[38] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. `http://archive.ics.uci.edu/ml`. 2017.

[39] Richard Duda, Peter Hart, and David G.Stork. *Pattern Classification*. 2001.

[40] Jonathan Eckstein, Peter L. Hammer, Ying Liu, Mikhail Nediak, and Bruno Simeone. “The Maximum Box Problem and its Application to Data Analysis”. In: *Computational Optimization and Applications* 23.3 (2002), pp. 285–298.

[41] Leon Eifler, Ambros Gleixner, Matthias Miltenberger, and Daniel Rehfeldt. *SoPlex - Sequential object-oriented simPlex*. `https://soplex.zib.de/`.

[42] Meriem El Karoui, Monica Hoyos-Flight, and Liz Fletcher. “Future Trends in Synthetic Biology—A Report”. In: *Frontiers in Bioengineering and Biotechnology* 7 (2019), p. 175.

[43] Chin-Yuan Fan, Pei-Chann Chang, Jyun-Jie Lin, and J.C. Hsieh. “A hybrid model combining case-based reasoning and fuzzy decision tree for medical data classification”. In: *Applied Soft Computing* 11.1 (2011), pp. 632 –644.

[44] Thalia Farazi, Hugo Horlings, Jelle ten Hoeve, et al. “MicroRNA Sequence and Expression Analysis in Breast Tumors by Deep Sequencing”. In: *Cancer research* 71 (2011), pp. 4443–53.

[45] Terrence Furey, Nello Cristianini, David Bednarski, and David Haussler. “Support Vector Machine Classification and Validation of Cancer Tissue Samples Using Microarray Expression Data”. In: *Bioinformatics* 16 (2001).

[46] Cristian Gallo, Rocío Cecchini, Jessica Carballido, Sandra Micheletto, and Ignacio Ponzoni. “Discretization of gene expression data revised”. In: *Briefings in bioinformatics* 17 (2015).

[47] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. “Abstract Gringo”. In: *Theory and Practice of Logic Programming* 15 (2015).

[48] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. *Potassco user guide*. `https://github.com/potassco/guide`. 2019.

[49] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. “Potassco: The Potsdam Answer Set Solving Collection”. In: *AI Commun.* 24.2 (2011), 107–124.

[50] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.

[51] Michael Gelfond and Vladimir Lifschitz. “The Stable Model Semantics for Logic Programming”. In: *Proceedings of International Logic Programming Conference and Symposium*. MIT Press, 1988, pp. 1070–1080.

[52] Ambros Gleixner, Michael Bastubbe, Leon Eifler, et al. *The SCIP Optimization Suite 6.0*. ZIB-Report 18-26. 2018.

[53] Ambros M. Gleixner, Daniel E. Steffy, and Kati Wolter. *Iterative Refinement for Linear Programming*. ZIB-Report 15-15. 2015.

[54] Cui Guo and Hong Seo Ryoo. “Compact MILP models for optimal and Pareto-optimal LAD patterns”. In: *Discrete Applied Mathematics* 160.16 (2012), pp. 2339 –2348.

[55] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. `http://www.gurobi.com`. 2019.

[56] Peter Hammer and Tiberius Bonates. “Logical analysis of data—An overview: From combinatorial optimization to medical applications”. In: *Annals of Operations Research* 148 (2006), pp. 203–225.

[57] Peter L. Hammer, Alexander Kogan, Bruno Simeone, and Sándor Szedmák. “Pareto-optimal patterns in logical analysis of data”. In: *Discrete Applied Mathematics* 144.1 (2004), pp. 79 –102.

[58] Henry H.Q. Heng, Joshua B. Stevens, Steven W. Bremer, Guo Liu, Batoul Y. Abdallah, and Christine J. Ye. “Evolutionary Mechanisms and Diversity in Cancer”. In: Advances in Cancer Research 112 (2011), pp. 217 –253.

[59] Juan Félix Ávila Herrera. “Mixed Integer Linear Programming Based Implementations of Logical Analysis of Data and Its Applications”. PhD thesis. 2013.

[60] Robert C. Holte, Liane Acker, and Bruce W. Porter. *Concept Learning and the Problem of Small Disjuncts*. 1989.

[61] Alex Kean and George Tsiknis. “An incremental method for generating prime implicants/implicates”. In: *Journal of Symbolic Computation* 9.2 (1990), pp. 185 –206.

[62] Zoltán Kis, Hugo Pereira, Takayuki Homma, Ryan Pedrigi, and Rob Krams. “Mammalian synthetic biology: Emerging medical applications”. In: *Journal of the Royal Society, Interface / the Royal Society* 12 (2015).

[63] Hiroaki Kitano. “Systems Biology: A Brief Overview”. In: *Science (New York, N.Y.)* 295 (2002), pp. 1662–4.

[64] Bertram Klinger, Anja Sieber, Raphaela Fritsche, et al. “Network quantification of EGFR signaling unveils potential for targeted combination therapy”. In: *Molecular systems biology* 9 (2013), p. 673.

[65] Thorsten Koch. “Rapid Mathematical Prototyping”. PhD thesis. Technische Universität Berlin, 2004.

[66] Sotiris Kotsiantis and D. Kanellopoulos. “Discretization techniques: A recent survey”. In: *GESTS International Transactions on Computer Science and Engineering* 32 (2005), pp. 47–58.

[67] Michael Lauer, Sorin Alexe, Claire Pothier, Eugene Blackstone, Hemant Ishwaran, and Peter L. Hammer. “Use of the Logical Analysis of Data Method for Assessing Long-Term Mortality Risk After Exercise Electrocardiography”. In: *Circulation* 106 (2002), pp. 685–90.

[68] Yuh-Jye Lee, Olvi L. Mangasarian, and William H. Wolberg. “Breast Cancer Survival and Chemotherapy: A Support Vector Machine Analysis”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 55 (2002).

[69] Miguel Lejeune, Vadim Lozin, Irina Lozina, Ahmed Ragab, and Soumaya Yacout. “Recent advances in the theory and practice of Logical Analysis of Data”. In: *European Journal of Operational Research* 275.1 (2019), pp. 1 –15.

[70] Pierre Lemaire. *Ladoscope*. `http://www.kamick.org/lemaire/software.html`.

[71] Gianpiero Di Leva and Carlo M Croce. “miRNA profiling of cancer”. In: *Current Opinion in Genetics and Development* 23.1 (2013), pp. 3 –11.

[72] Vladimir Lifschitz. “Twelve Definitions of a Stable Model”. In: *Lecture Notes in Computer Science* (2008). Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP’08), 37–51.

[73] Vladimir Lifschitz. “What is Answer Set Programming?” In: AAAI’08 (2008), 1594–1597.

[74] Kazuhisa Makino, Takashi Suda, Kojin Yano, and Toshihide Ibaraki. “Data analysis by positive decision trees”. In: *IEICE Transactions on Information and Systems* E82-D, No.1 (1999), pp. 76–88.

[75] Olvi L. Mangasarian, W. Nick Street, and William H. Wolberg. “Breast Cancer Diagnosis and Prognosis Via Linear Programming”. In: *Operations Research* (1995), pp. 548–725.

[76] Victor W. Marek and Miroslaw Truszczynski. “Stable models and an alternative logic programming paradigm”. In: *CoRR* (1998).

[77] Eddy Mayoraz. *C++ tools for logical analysis of data*. `http://rutcor.rutgers.edu/pub/LAD/man.pdf`. 1998.

[78] Pejman Mohammadi, Niko Beerenwinkel, and Yaakov Benenson. “Automated Design of Synthetic Cell Classifier Circuits Using a Two-Step Optimization Strategy.” In: *Cell systems* 4 2 (2017), pp. 207–218.

[79] Tae Seok Moon, Chunbo Lou, Alvin Tamsir, Brynne Stanton, and Christopher Voigt. “Genetic Programs Constructed from Layered Logic Gates in Single Cells”. In: *Nature* 491 (2012).

[80] Ilkka Niemelä, Patrik Simons, and Timo Soininen. “Stable Model Semantics of Weight Constraint Rules”. In: *LPNMR*. 1999.

[81] Ilkka Niemelä. “Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm”. In: *Ann. Math. Artif. Intell.* 25 (1999), pp. 241–273.

[82] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. “Smodels: A System for Answer Set Programming”. In: *CoRR* cs.AI/0003033 (2000).

[83] Luigi Palopoli, Fiora Pirri, and Clara Pizzuti. “Algorithms for selective enumeration of prime implicants”. In: *Artificial Intelligence* 111.1 (1999), pp. 41 –72.

[84] Yuliang Pan, Jun Zhang, Huiqun Fu, and Liangfang Shen. “miR-144 functions as a tumor suppressor in breast cancer through inhibiting ZEB1/2-mediated epithelial mesenchymal transition process”. In: *OncoTargets and Therapy* 9 (2016), pp. 6247–6255.

[85] Vili Podgorelec, Peter Kokol, Bruno Stiglic, and Ivan Rozman. “Decision Trees: An Overview and Their Use in Medicine”. In: *J. Med. Syst.* 26.5 (2002), pp. 445–463.

[86] Willard V. Quine. “The problem of simplifying truth functions”. In: *The American Mathematical Monthly* 59.8 (1952), 521–531.

[87] J. Ross Quinlan. “Induction of Decision Trees”. In: *Mach. Learn.* 1.1 (1986), pp. 81–106.

[88] Anupama Reddy, Honghui Wang, Hua Yu, Tiberius Bonates, Vimla Gulabani, Joseph Azok, Gerard Hoehn, Peter L Hammer, Alison E Baird, and King Li. "Logical Analysis of Data (LAD) model for the early diagnosis of acute ischemic stroke". In: *BMC medical informatics and decision making* 8 (2008), p. 30.

[89] UC Irvine Machine Learning Respository. *Breast Cancer Wisconsin (Original) Data Set*. https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original). University of Wisconsin Hospitals, Madison: Dr. William H. Wolberg.

[90] UC Irvine Machine Learning Respository. *Heart Disease Data Set*. https://archive.ics.uci.edu/ml/datasets/Heart+Disease. Hungarian Institute of Cardiology. Budapest: Andras Janosi, M.D, University Hospital, Zurich, Switzerland: William Steinbrunn, M.D, University Hospital, Basel, Switzerland: Matthias Pfisterer, M.D., V.A. Medical Center, Long Beach and Cleveland Clinic Foundation:Robert Detrano, M.D., Ph.D.

[91] Giuseppina Roscigno, Ilaria Puoti, Immacolata Giordano, et al. "MiR-24 induces chemotherapy resistance and hypoxic advantage in breast cancer". In: *Oncotarget* 8 (2017).

[92] Frank Rosenblatt. *The perceptron: a theory of statistical separability in cognitive systems (Project Para)*. Cornell Aeronautical Laboratory, 1958.

[93] Hong Seo Ryoo and In-Yong Jang. "MILP approach to pattern generation in logical analysis of data". In: *Discrete Applied Mathematics* 157.4 (2009), pp. 749 –761.

[94] Yvan Saeys, Iñaki Inza, and Pedro Larranaga. "A review of feature selection techniques in bioinformatics". In: *Bioinformatics (Oxford, England)* 23 (2007), pp. 2507–17.

[95] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.

[96] Patrik Simons, Ilkka Niemelä, and Timo Soininen. "Extending and implementing the stable model semantics". In: *Artificial Intelligence* 138.1 (2002). Knowledge Representation and Logic Programming, pp. 181 –234.

[97] Vijai Singh. "Recent advances and opportunities in synthetic logic gates engineering in living cells". In: *Systems and Synthetic Biology* 8 (2014), 271–282.

[98] Piro Siuti, John Yazbek, and Timothy Lu. "Synthetic circuits integrating logic and memory in living cells". In: *Nature biotechnology* 31 (2013).

[99] Shimyn Slomovic, Keith Pardee, and James Collins. “Synthetic biology devices for in vitro and in vivo diagnostics”. In: *Proceedings of the National Academy of Sciences* 112 (2015).

[100] IBM ILOG CPLEX Optimization Studio. *CPLEX Optimizer*. `https://www.ibm.com/analytics/cplex-optimizer`. 2019.

[101] SCIP Optimization Suite. *SCIP manual*. `https://scip.zib.de/doc-6.0.0/html/`.

[102] Jonathan Teo, Sung Sik Woo, and Rahul Sarpeshkar. “Synthetic Biology: A Unifying View and Review Using Analog Circuits”. In: *IEEE transactions on biomedical circuits and systems* 9 (2015).

[103] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[104] Zhen Xie, Liliana Wroblewska, Laura Prochazka, Ron Weiss, and Yaakov Benenson. “Multi-Input RNAi-Based Logic Circuit for Identification of Specific Cancer Cells”. In: *Science (New York, N.Y.)* 333 (2011), pp. 1307–11.

[105] Kedong Yan and Hong Ryoo. “A multi-term, polyhedral relaxation of a 0–1 multilinear function for Boolean logical pattern generation”. In: *Journal of Global Optimization* 74 (2018).

[106] Kedong Yan and Hong Seo Ryoo. “0-1 multilinear programming as a unifying theory for LAD pattern generation”. In: *Discrete Applied Mathematics* 218 (2017), pp. 21 –39.

[107] Kedong Yan and Hong Seo Ryoo. “Strong valid inequalities for Boolean logical pattern generation”. In: *Journal of Global Optimization* 69.1 (2017), pp. 183–230.

[108] *ZIMPL - Zuse Institut Mathematical Programming Language*. `https://zimpl.zib.de/`.

Danksagung

Zuerst möchte ich mich bei meinem Doktorvater Alexander Bockmayr bedanken. Danke für Deine Unterstützung und Deine wissenschaftlichen Anstöße in den letzten Jahren, für die Klarheit Deiner Gedanken und Formulierungen. Neben all dem, danke ich Dir dafür, dass Du auch für nicht arbeitsbezogene Themen stets ein offenes Ohr hattest und immer klar erkannt hast, wenn das normale Leben gerade wichtiger und zeiteinnehmender war als die wissenschaftliche Arbeit.

Mein zweiter Dank gilt Torsten Schaub. Nachdem Alexander mich auf ASP gestoßen hat, waren es vor allem die Unterhaltungen, Vorlesungen und Vorträge in Potsdam, die mich von der Schönheit von ASP überzeugt haben. Es macht Spaß zu sehen, wenn jemand so für ein Thema brennt und das auch weitergeben kann. Danke, dass Ihr mich in Potsdam immer mit offenen Armen und Ohren empfangen habt und Du schließlich auch diese Arbeit als Zweitgutachter übernommen hast.

Ein Großteil meiner Arbeitszeit wurde über das CSB, das Graduiertenkolleg *Computational Systems Biology*, finanziert. An dieser Stelle möchte ich mich für diese Hilfe bedanken. Neben dem Finanziellen war das CSB aber auch immer eine Anlaufstelle für wissenschaftlichen Austausch und ein Ausgangspunkt für Kooperationen und Freundschaften. Hier bedanke ich mich vor allem bei Cordelia für viel Organisation und Hilfsbereitschaft. Außerdem möchte ich Nils Blüthgen, meinem Zweitbetreuer, danken. Gerade am Anfang war mir Deine Arbeit ein spannender Anstoß. Danke auch dafür, dass Du und Alexander die Freundschaft (und wissenschaftliche Kooperation) von Torsten und mir schon geplant hattet. Es hat funktioniert. Danke, Torsten, für die schöne Zeit. Es ist eine wahre Bereicherung sich mit jemandem sowohl wissenschaftlich als auch menschlich so gut zu verstehen.

Ich möchte mich auch bei den Arbeitsgruppen *Mathematics in Life Sciences* und *Discrete Biomathematics* um Alexander Bockmayr und Heike Siebert bedanken. Von Anfang an habe ich mich unter Euch allen sehr wohl gefühlt. Heike, auch wenn Du offiziell nie für diese Rolle vorgesehen warst, habe ich mich von Dir immer gut betreut gefühlt. Danke für die Unterhaltungen zu Mathe und der Welt und die Hinweise zu meiner Arbeit. Vielen Dank auch an Katja. Du bist stets bereit uns alle zu unterstützen und findest immer eine Lösung. Im Besonderen möchte ich noch Dir, Kirsten, danken. Wir haben die längste Zeit zusammen in einem Büro verbracht. Auch wenn wir dies nun schon eine Weile nicht mehr tun, so bleibt unsere Freundschaft hoffentlich noch lange bestehen. Für die letzten Monate dieser Arbeit bin ich doch noch ins richtige Mathegebäude eingezogen. Lin und ich haben gemeinsam diesen großen Schritt des Aufschreibens zu Ende gebracht. Thank you, Lin, for being in the same situation with me, for your good tea and for your empowering words!

Natürlich wird solch eine Arbeit geschrieben um gelesen zu werden. Es gibt ein paar Menschen, die das in Teilen auf jeden Fall schon getan haben. Dafür möchte ich euch herzlich danken! Danke Alexander, André, Kai, Kirsten, Philipp und Sarah!

Zuletzt möchte ich von Herzen meiner Familie danken. Danke, Papa, dass Du mir von Anfang an alles ganz genau erklärt und an mich geglaubt hast. Danke, Mama, für all Deine Unterstützung und den Rückhalt auf dem Weg hierher.

Moritz, ohne Dich hätte mir diese Zeit nicht halb so viel Spaß gemacht. Mit Dir ist das Leben schön. Danke, Mira. Du bist die pure und schönste Ablenkung. Und danke an den kleinen Bruder, dass Du mich das Ganze doch noch hast zu Ende bringen lassen.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich alle Hilfsmittel und Hilfen angegeben habe und versichere, auf dieser Grundlage die Arbeit selbständig verfasst zu haben. Die Arbeit wurde nicht schon einmal in einem früheren Promotionsverfahren eingereicht.

Berlin, Januar 2020

Katinka Becker