Annals of Mathematics and Artificial Intelligence (2025) 94:63–87
https://doi.org/10.1007/s10472-025-09981-x

RESEARCH

Answer set programming for pattern generation in logical analysis of data

Katinka Becker[1,2] · Alexander Bockmayr[2]

Accepted: 2 April 2025 / Published online: 16 April 2025
© The Author(s) 2025

Abstract
Logical Analysis of Data (LAD) is a powerful technique for data classification based on partially defined Boolean functions. The decision rules for class prediction in LAD are formed out of patterns. According to different preferences in the classification problem, various pattern types have been defined. The generation of these patterns plays a key role in the LAD methodology and represents a computationally hard problem. In this article, we introduce a new approach to pattern generation in LAD based on Answer Set Programming (ASP), which can be applied to all common LAD pattern types.

Keywords Answer set programming · Logical analysis of data · Pattern generation · Classification

1 Introduction

Logical Analysis of Data (LAD) is a data analysis methodology that combines ideas from combinatorics and Boolean functions with machine learning and optimization. It was originally introduced by Peter L. Hammer in a lecture in 1986 and later extended and published together with Yves Crama and Toshihide Ibaraki in [1]. Subsequently, the method has been further developed over the last decades [2–4] and applied to various research fields, see e.g. [5–12].

The data analysis process with LAD consists of three main steps: discretization, pattern generation and theory formation [1, 2]. One of the key advantages of LAD over other well-known machine learning methods, such as support vector machines or neural networks (see e.g. [13]), is the generation of patterns. Patterns are the fundamental concept of LAD. They extract important information from the given data that is easier to interpret and, therefore, easier to communicate than, e.g., separating hyperplanes resulting from classification via a support vector machine, or weights in a neural network obtained by backpropagation. Patterns

✉ Katinka Becker
katinka.becker@ptb.de

Alexander Bockmayr
alexander.bockmayr@fu-berlin.de

1 Division Medical Physics and Metrological Information Technology, Physikalisch-Technische Bundesanstalt, Abbestr. 2-12, Berlin 10587, Germany

2 Department of Mathematics and Informatics, Freie Universität Berlin, Arnimallee 6, Berlin 14195, Germany

provide understandable information that can be used directly to justify the results on the given data [14], which adds to the explainability of the models.

In this article, we focus on the generation of patterns, which is the central task of LAD. Various types and properties of patterns have been considered in the literature and their relative efficiency has been analyzed [15]. The generation of LAD patterns is computationally hard because their number may be exponential in the size of the input. Most common approaches for pattern enumeration are based on Mixed-Integer Linear Programming (MILP) [16–22].

Here, we present a novel approach to pattern generation, which uses Answer Set Programming (ASP) [23–27]. ASP is a declarative programming paradigm oriented towards difficult, primarily NP-hard, search and combinatorial optimization problems. It is therefore perfectly suited to the given tasks in pattern generation. Using the declarative nature of ASP and the power of ASP solvers, we are able to compute all common LAD pattern types in a modular systematic and efficient way, see Fig. 2 and the benchmarks in Tables 2 and 3. Our ASP approach can outperform state-of-the-art MILP models as shown in Sect. 5 for the class of maximal patterns. The modular structure of the ASP encoding and its clear and succinct presentation allow the programs to be easily adapted to different tasks or pattern types. In particular, ASP provides a highly flexible environment to narrow down the search to specific patterns of interest. For practical applications, where the number of patterns is typically huge, this is of utmost importance.

The paper is organized as follows. In Sect. 2, we summarize basic concepts and notation of LAD, followed in Sect. 3 by an outline of the syntax and semantics of ASP. In Sect. 4, we develop our ASP approach to generating the different pattern types in LAD. To illustrate the performance of our approach, computational results are given in Sect. 5, including a comparison with MILP [16]. In addition to computational efficiency, flexibility is another major benefit of the ASP approach. We demonstrate this in Sect. 6, where we describe how our programs for pattern generation can be adapted to specific user requirements. Finally, in Sect. 7, we present the conclusions of our work.

The method presented here was first developed in [28]. The ASP approach for generating prime patterns in Sect. 4.3 was originally published in [29].

2 Logical analysis of data

Let $\Omega \subseteq \{0,1\}^n$ be a set of *observations* that is divided into two subsets by an additional *decision variable* x_0, such that $\Omega = \Omega^+ \uplus \Omega^-$ is the disjoint union of Ω^+ and Ω^-, called *positive* resp. *negative observations* (see Table 1).

A *Boolean function* of n variables is a mapping $f\colon \{0,1\}^n \rightarrow \{0,1\}, n \geq 1$. A data set Ω as described above leads to a *partially defined Boolean function* (pdBf) (Ω^+, Ω^-),

Table 1 Binary data set Ω partitioned into positive observations Ω^+ and negative observations Ω^- by decision variable x_0

#	x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	1	0	0	1	0	1	1
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
6	0	1	1	0	0	0	1
7	0	1	0	1	0	1	1

meaning that the function f is only defined on the subset $\Omega = \Omega^+ \uplus \Omega^- \subseteq \{0, 1\}^n$, with $f(x) = 1$ if $x \in \Omega^+$ and $f(x) = 0$ if $x \in \Omega^-$. Every Boolean function $e: \{0, 1\}^n \to \{0, 1\}$ with $e(x) = f(x)$ for all $x \in \Omega$ is called an *extension* of f. A major goal of LAD is to find an extension e of a given pdBf (Ω^+, Ω^-) that can be used to classify new data points $y \in \{0, 1\}^n \setminus \Omega$ as positive or negative.

2.1 Patterns and pattern types

The extensions of a pdBf, called *theories* in the LAD methodology, are built out of *patterns*, which play a key role. Next we introduce the concept of positive and negative patterns and define various pattern types.

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. The *degree* of a term is the number of its 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 $t(v) = 1$.

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

In general, the number of patterns is huge compared to the size of the data set. To gain more clarity in this large amount of patterns, it is not only convenient but necessary to define properties for patterns that allow to rank them according to certain criteria. Various properties and types of patterns have been studied and their relative efficiency has been analyzed [15].

Let P be a pattern of the pdBF (Ω^+, Ω^-) in n variables. 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.

The pattern P is called

- *prime* if $Lit(P)$ is an inclusionwise minimal set of literals, i.e., there is no pattern P' such that $Lit(P') \subsetneq Lit(P)$.
- *strong* if it has an inclusionwise maximal coverage, i.e., there is no pattern P' such that $Cov(P) \subsetneq Cov(P')$.
- *spanned* if it does not strictly include any other Boolean subcube containing the same set of observations.
- *strong prime* if it is both strong and prime.
- *strong spanned* if it is both strong and spanned.
- *maximal* if it is maximal with respect to the number of observations covered, i.e., there is no pattern P' such that $|Cov(P)| < |Cov(P')|$.

2.2 Preferences

An alternative way of introducing the different pattern types is based on pattern preferences [15]. Given two patterns P and P' of a pdBF (Ω^+, Ω^-) we say that

- P is *simplicity-wise preferred* to P', written $P' \leq_\sigma P$, if the set of literals in P is contained in the set of literals in P', i.e., $Lit(P) \subseteq Lit(P')$.

- P is *selectivity-wise preferred* to P', written $P' \leq_{\Sigma} P$, if the subcube of pattern P is included in the subcube of pattern P', i.e., $S(P) \subseteq S(P')$.
- P is *evidentially preferred* to P', written $P' \leq_{\epsilon} P$, if the set of observations covered by P includes the observations covered by P', i.e. $Cov(P') \subseteq Cov(P)$.

A pattern P is called *Pareto-optimal* with respect to the preference $\rho \in \{\sigma, \Sigma, \epsilon\}$ if there is no pattern $P' \neq P$ with $P \leq_{\rho} P'$. The simultaneous satisfaction of $P \leq_{\rho} P'$ and $P' \leq_{\rho} P$ is denoted by $P \approx_{\rho} P'$ for any preference ρ.

Two preferences ϕ and ρ can be combined by intersection and lexicographic refinement in the following way:

- A pattern P is preferred to a pattern P' with respect to the *intersection* $\phi \wedge \rho$ if and only if $P' \leq_{\phi} P$ and $P' \leq_{\rho} P$.
- A pattern P is preferred to a pattern P' with respect to the *lexicographic refinement* $\phi \mid \rho$ if and only if either $P' <_{\phi} P$ or $P \approx_{\phi} P'$ and $P' \leq_{\rho} P$.

It follows that a pattern P of a pdBF (Ω^{+}, Ω^{-}) is

- prime iff P is Pareto-optimal with respect to σ,
- strong iff P is Pareto-optimal with respect to ϵ,
- spanned iff P is Pareto-optimal with respect to $\Sigma \wedge \epsilon$,
- strong prime iff P is Pareto-optimal with respect to $\epsilon \mid \sigma$,
- strong spanned iff P is Pareto-optimal with respect to $\epsilon \mid \Sigma$.

2.3 Pattern parameters

To measure the performance of a pattern on a given data set, two key parameters have been introduced: homogeneity and prevalence [30, 31].

The *homogeneity* $Hom^{+}(P)$ of a positive pattern P is given by

$$Hom^{+}(P) = \frac{|Cov(P)|}{|Cov_{total}(P)|}, \tag{1}$$

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. 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).

The *prevalence* $Prev^{+}(P)$ of a positive pattern P is given by

$$Prev^{+}(P) = \frac{|Cov(P)|}{|\Omega^{+}|}. \tag{2}$$

The prevalence $Prev^{-}(P)$ of a negative pattern P is defined analogously. A positive pattern has a high positive prevalence if it covers a large amount of the positive observations.

3 Answer set programming

Answer Set Programming (ASP) [23–27] is a declarative programming paradigm. In contrast to imperative programming, a program consists of a high-level description of *what the problem is* rather than a sequence of instructions *how the problem should be solved*. In ASP, the

problem is specified by a set of logical rules, which are first instantiated by an ASP *grounder* and then solved by an ASP *solver*. ASP is oriented towards difficult, primarily NP-hard, search problems. Various systems for grounding and solving have been developed, see for example [32–34]. Here we use the ASP system `clingo`, a combination of the grounder `gringo` and the solver `clasp` developed by Potassco[1], the Potsdam Answer Set Solving Collection [34, 35].

ASP is based on the stable model semantics of logic programming [36, 37]. Search problems are reduced to the computation of stable models or answer sets, which are found by ASP solvers. A program in ASP consists of a finite set of logical rules of the form

$$A_0 \text{ :- } A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n. \tag{3}$$

Here $0 \leq m \leq n$ and each A_i, $1 \leq i \leq n$, is an *atom*, i.e., an atomic formula in first-order logic, and 'not' stands for negation by default [36]. The left-hand side of the rule is called *head* and the right-hand side of the rule is called *body*. The intuitive meaning of (3) is the logical implication stating that the head is true if the body is true. In the special case $n = 0$, rule (3) is called a *fact* and denoted by

$$A_0. \tag{4}$$

Such a fact expresses that the atom A_0 is always true.

Omitting A_0 in (3) amounts to taking A_0 to be false, and rule (3) represents an *integrity constraint*. Accordingly, the resulting rule

$$\text{:- } A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n. \tag{5}$$

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

To facilitate the use of ASP in practice, several extensions have been developed [38]. *Conditional literals* are of the form

$$A : B_1, \ldots, B_m \tag{6}$$

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

$$s \; \{C_1; \ldots; C_n\} \; t \tag{7}$$

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 $s, t \in \mathbb{N}$.

In addition to cardinality constraints (using #count for the number of elements) the language of ASP provides further aggregates such as #sum (the sum of weights; used for expressing weight constraints), #min (the minimum weight), and #max (the maximum weight) [39].

Optimization problems can also be formulated in ASP. Objective functions minimizing the sum of weights w_i of literals B_i are expressed as

$$\texttt{\#minimize} \; \{w_1 : B_1; \ldots; w_n : B_n\} \tag{8}$$

[1] https://potassco.org/

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. More formally, 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. A set X of atoms is a *stable model* of a program P if the inclusionwise minimal model of the reduct P^X of P relative to X is equal to X.

For a comprehensive description of ASP and the software `clingo` we refer to [27, 39]. All calculations in this paper were done using `clingo 5.4.0`.

4 Pattern generation in ASP

ASP is particularly well-suited for combinatorial (optimization) problems, and thus a natural choice as a framework for pattern generation in LAD. In this section, we explain how to use ASP as a new computational approach for efficient pattern calculation and enumeration.

4.1 Input data

To make use of ASP, the input data has to be provided in a suitable format. A binary `csv`-file containing one row for each observation, with the first column indicating whether the observation is negative or positive (represented by 0 or 1, respectively), can be translated to a corresponding input file for ASP, as demonstrated in the example in Fig. 1.

Each binary value of the input data is represented in ASP by a fact `i(Sign,ID, Variable,Value)`, where `Sign` is the sign of the observation (1 for positive and 0 for negative), `ID` is the identifier of the observation, `Variable` is the number of the Boolean variable and `Value` the binary value.

```
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. 1 Example input data set in `csv`-format and corresponding input data for ASP pattern generation

4.2 ASP programs for the common pattern types

Once the input data has been translated, the pattern generation process can be started. In the following, we develop ASP approaches to calculating different types of patterns. Besides *general* patterns without any specified preference, we will consider *prime*, *spanned*, *strong*, *maximal*, *strong prime* and *strong spanned* patterns.

The ASP programs can be divided into two classes, see Fig. 2. While general, prime and maximal patterns are obtained by *literal-based pattern generation*, strong and spanned patterns rely on *coverage-based pattern generation*. This division is due to the pattern-specific preferences that are optimized. Prime patterns have an inclusionwise minimal set of literals. This naturally leads to a program including rules to choose a suitable set of literals (in the LAD sense). In contrast, strong and spanned patterns are defined with respect to their coverage. For this reason, we base the programs on the set of covered observations.

4.3 Literal-based pattern generation

We first describe the ASP encodings for literal-based pattern generation.

4.3.1 General patterns

The ASP code for general pattern generation is given in Fig. 3. Note that it involves only 10 lines (without the 3 comment lines starting with `%`). This basic ASP program calculates all patterns, i.e., without any specific preference. By adding further lines to narrow down the answer set of this program, maximal and prime patterns can be obtained.

Before running the program, the user has to define the four constants `degree`, `sign`, `homogeneity` and `prevalence`, such that any stable model will define a pattern of the given degree and sign, with homogeneity and prevalence greater than or equal to the chosen constants.

The program is organized in three parts `GENERATE`, `DEFINE` and `TEST`. Any stable model should define a pattern of the given degree. Thus it has to include `degree` many atoms of the form `pat(S,B)`, where `S` is the name of a variable and `B` its Boolean value.

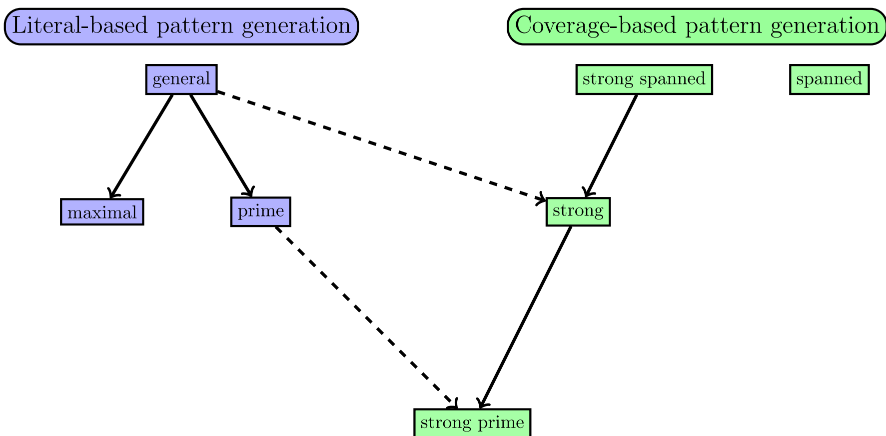

Fig. 2 Schematic overview of the hierarchy of the encodings for ASP pattern generation

```
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 Basic ASP encoding for general pattern generation

The rule in line 2 is a choice rule generating solution candidates. From the definition of a pattern we know 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 guaranteed that the chosen 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 down the stable model 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 in line 6 the predicate `covered(W,X)`, which is true if observation `X` having sign `W` is covered by the chosen set of literals. For defining this predicate, we use in line 5 the auxiliary predicate `not_covered(W,X)`, which is true for an observation `X`, if one of the literals `pat(S,B)` does not cover `X`. 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 predicates and test whether the choice of literals fulfills the definition of a pattern with given homogeneity and prevalence. Here, the constants `homogeneity` and `prevalence` have to be specified as integer percentages, with values between 0 and 100. A stable model then provides a pattern with homogeneity and prevalence greater than or equal to the chosen constants divided by 100. 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. The sum in line 10 and 11 follows directly from (1) in Sect. 2.3 if all terms are brought to one side. Every covered observation `X` of the correct sign `W=sign` (resp. the opposite sign `W!=sign`) contributes a weight `(homogeneity - 100)` (resp. `homogeneity`) to the sum. In addition, line 12 and 13 make sure that the prevalence condition is met by the generated pattern. Namely, the constant `prevalence` times the number of observations of the correct sign is greater than 100 times the number of observations of correct sign that are covered by the pattern. The unary predicate `nbrcorrectobs` is calculated in advance and counts the number of observations having the same sign as the desired pattern.

Before we continue, we briefly discuss the role of the input parameters. The constant `sign` specifies whether we compute positive or negative patterns. The generation of positive and negative patterns is symmetric. The two resulting pattern sets are used in opposite

ways, namely for the characterization of the set of positive resp. negative observations. 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 may become so large that it is no longer manageable. Therefore, it may be useful to include these parameters in advance. 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.

The constant `degree` was added for a similar reason. As the pool of answer sets gets large, it is helpful to specify the answer set one is looking for. If one wants to calculate all patterns of any degree, this constant can be left out from the input. Then line 2 in Fig. 3 gets replaced by the choice rule `1{pat(S,B):i(sign,_,S,B)}.` The degree of the pattern can be determined in a later step if needed.

4.3.2 Prime patterns

By expanding the encoding in Fig. 3 by just one more test, we can constrain the set of patterns to those that are prime. The additional lines in Fig. 4 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 Fig. 3 (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 fewer observations than the original pattern. A pattern that has a homogeneity above the given constant after deleting a literal is not a prime pattern. Further note that if the constant `degree` was left out from the input in the basic encoding, it has to be reintroduced here in an additional line occurring before line 3.

4.3.3 Maximal patterns

The calculation of maximal patterns is realized by adding just one line to the basic encoding in Fig. 3. Maximal patterns are patterns that have a maximal number of covered observations. Thus we can simply use the following statement at the end of the encoding:

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

```
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. 4 Additional lines to the encoding in Fig. 3 for the generation of prime patterns

4.4 Coverage-based pattern generation

While the computation of the three pattern types in Sect. 4.3 is based on stable models consisting of LAD literals, we will now use stable models consisting of covered observations. This follows naturally from the definition of spanned and strong patterns. Other than the literal-based encodings, the coverage-based encodings will generate patterns of any degree, homogeneity and prevalence (except for strong and strong prime patterns). This means that the constants degree, homogeneity and prevalence do not have to be specified by the user. The only parameter to be given in advance is the sign of the patterns. As discussed before, this can easily be modified by constraining the choice rules to a size of degree and adding two integrity constraints for homogeneity and prevalence.

4.4.1 Spanned patterns

In Fig. 5, the encoding for the generation of spanned patterns is shown. 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 in line 2 of the GENERATE part 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.

The DEFINE part is used to narrow down the chosen set to an accurate answer, namely a set of covered observations with the corresponding spanned pattern. Sets of pattern literals able to cover the chosen set of observations are generated using the following rules. In line 6, the predicate lit_candidate(S,B) is satisfied by each literal that covers a chosen 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

```
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. 5 Encoding for the generation of spanned patterns

can be included in a pattern. For this reason, 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* of the form `not_lit(S,B)`. Here, a *not literal* 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 this is not the case, and that the literal combination is indeed a pattern, the rules in lines 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 covers. The number `D` of literals covering an observation `Y` of opposite sign is 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 the literals of the generated term (line 21). In this case, the term is a selectivity-wise preferred pattern.

For the evidential preference of the pattern, we use the rules in lines 15-18. We have to make sure that the chosen set of covered observations is as large as possible with respect to the set of literals. Therefore, we collect all observations of the correct sign that 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 constituted by the set of literals is not evidentially preferred and thus cannot be spanned. Therefore, we exclude these results from the set of answers (line 22). A stable model of this program is a spanned pattern together with the set of observations that it covers.

4.4.2 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. These are the strong patterns having an inclusionwise maximal set of literals. For this reason, the structure of the encoding is similar to the one used for calculating spanned patterns. The whole program is shown in Fig. 6. Here we explain the parts in which this program differs from the previous one.

The first 13 lines are identical to those in Fig. 5. A set of covered observations is chosen, and the set of literals of the pattern is defined by the set of literals covering all the chosen observations. In 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. To this end, we first define the predicate `lit_in_new(Y,(S,B))` to determine which of the literals `(S,B)` that have already been selected also cover an observation `Y` of correct sign that is not included in the set of chosen covered observations (line 15). We use the same predicate structure as before for the literals `lit(S,B)` to test whether this subset of literals also covers an observation of opposite sign. In detail, we define the predicate `litinnew_in_op(Z,(S,B),Y)` for every pair of observations `Z` of the opposite sign and `Y` of the correct sign that is not in the originally chosen set of observations. This predicate is true if the literal `(S,B)` covers the observation `Z` (line 19). In the following lines 20-22, we count the `D` literals that cover the observation `Z` of opposite sign. Then we compare this number in line 23 to the full number of literals that cover all chosen observations plus the observation `Y`. If these two numbers are the same, i.e., the whole term covers an observation

```
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. 6 Encoding for the generation of strong spanned patterns

of opposite sign, then the term is not a pattern. This implies that we do not find a pattern that covers the chosen set of observations plus the observation `Y`. In this 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 a 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 that covers them all (line 27). The resulting answer sets are all strong spanned patterns.

4.4.3 Strong patterns

The encoding for strong patterns is based on how they are defined in relation to strong spanned patterns. The additional lines of code are shown in Fig. 7. The property *spanned* ensures that a stable model for the strong spanned pattern calculation in Fig. 6 consists of an inclusionwise maximal set of literals. The full set of strong patterns contains all possible combinations of literals that form a pattern for an inclusionwise maximal set of covered observations. Therefore, we can generate the set of strong patterns from the set of strong spanned patterns by combining the computed 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 bounds on homogeneity and prevalence). The additional lines in Fig. 7 are the same as the encoding for general patterns in Fig. 3, except for line 2, where

```
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. 7 Additional lines to the encoding in Fig. 6 for the generation of strong patterns

the predicate `pat(S,B)` is chosen out of the already calculated set of literals, instead of all possible literals for a given data set.

4.4.4 Strong prime patterns

The encoding for generating strong prime patterns is based on the calculation of strong patterns, as the strong prime patterns form a subset of the strong patterns. We show the additional lines to the encoding of Fig. 7 in Fig. 8. They are very similar to the code for generating prime patterns in Fig. 4. The only difference is that in line 2 the degree `D` has to be computed and stored in the predicate `countdegree(D)`, as it is not given to the program as a constant.

4.5 Optimal patterns with `asprin`

In Sect. 4.3 and 4.4 we described in detail our ASP approach to generating the various Pareto-optimal pattern types defined by Hammer et al. [15]. For each pattern type, we presented

```
1  % DEFINE
2  countdegree(D) :- D=#sum{ 1,(S,B) : pat(S,B) }.
3
4  covered_after_deletion(W,X,(S,B)) :- i(W,X,_,_), pat(S,B),
5          not i(W,X,S,B), #sum{ 1,(T,C) : pat(T,C), i(W,X,T,C),
6          (T,C)!=(S,B) }= D-1, countdegree(D).
7
8  homgood(S,B) :- pat(S,B), #sum{ homogeneity-100,X:
9          covered_after_deletion(W,X,(S,B)) ,W=sign;
10         homogeneity,X : covered_after_deletion(W,X,(S,B)), W!=sign }
11         <= 0.
12
13 % TEST
14 :- homgood(S,B).
```

Fig. 8 Additional lines to the encoding in Fig. 7 for the generation of strong prime patterns

an ASP formulation encoding the desired preference via hard constraints. These programs can then be executed by an ASP grounder and solver. We used `clingo` [34] for all our experiments, but various other systems for ASP grounding and solving are also available. The Potsdam Answer Set Solving Collection Potassco [27] comprises not only `clingo`, but also several other ASP systems, which have been developed for specific purposes. One such system is `asprin` [40, 41]. While `clingo` includes functionalities for single objective and lexicographic optimization with priorities or weights, the `asprin` framework allows for the computation of optimal answer sets with preferences.

A *preference relation* in `asprin` is a strict partial order ρ over the stable models of a logic program. Given two stable models X and Y, the ordering $X <_\rho Y$ means that Y is preferred to X with respect to ρ. A stable model Y is optimal with respect to ρ if there is no other stable model X with $Y <_\rho X$. A preference statement in the program consists of an identifier, a type, and a set of preference elements. The identifier gives the name of the preference relation, the type and the set of elements define the relation. Some basic preference types like subset minimality and Pareto-optimality are predefined in the `asprin` library. Besides that, new preferences can be defined by the user. In the following, we give a short overview of the functionalities of `asprin` needed in our context. For a more detailed description we refer to [39–41].

Using `asprin`, the different LAD pattern types can easily be implemented based on the encoding for general pattern generation in Fig. 3, varying only the preference specifications at the end of the program. More precisely, the three preferences for LAD patterns (see Sect. 2.2) can be implemented in the following way.

- Simplicity preference:

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

- Selectivity preference:

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

- 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. For example, we define the preference `simplicity` of type `subset` over the atoms of the predicate `pat/2`. To specify the optimization objective, a directive `#optimize(P)` has to be added at the end, which tells `asprin` to compute answer sets that are optimal with respect to the preference `P`.

To generate *prime patterns* with `asprin`, we expand the encoding for general pattern generation in Fig. 3 by the following lines:

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

To calculate *strong patterns* the program is modified by adding:

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

In the `asprin` framework, preferences can also be combined using different principles. Here we use two such principles, namely `pareto` and `lexico`. The former refers to Pareto-optimality, while the latter pertains to optimality with respect to lexicographic refinement. For generating *spanned patterns* we write:

```
#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. 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).
```

5 Computational results

To illustrate the efficiency of our approach, we performed various computational experiments. In Sect. 5.1, we compare the running times for the computation of all patterns of a given type with and without `asprin` on six data sets that are widely used in LAD research. In Sect. 5.2, we report on the performance of the `asprin` approach for larger data sets with several hundreds of observations and features. Finally, in Sect. 5.3, we present a comparative study on the running time of our ASP approach for maximal pattern generation and a Mixed-Integer Linear Program (MILP) for the same purpose [16].

All calculations were made on a Linux AMD64 with 3.20 GHz, 4 cores and 15.6 GB of memory. The ASP programs were solved using the solver `clingo 5.4.0` [34]. The MILP problems in Sect. 5.3 were solved using the `SCIP` Optimization Suite 9.0.0 with the LP solver `SoPlex` [42, 43].

5.1 Comparing `asprin` and hard constraints

In Sect. 4.5 we showed how `asprin`, which is based on preferences in LAD, can be used for a succinct implementation of the different pattern types. Regarding simplicity and readability, using `asprin` is definitely preferable to the encodings presented in Sect. 4.3 and 4.4. In order to compare the running time of the two approaches, we performed tests on six widely

used data sets from the UC Irvine Machine Learning Repository [44]. These data sets, which appear frequently in the field of classification research and especially in the LAD literature, are *Breast Cancer Wisconsin (Original)* (BCW), *Heart Disease* (HD), *BUPA Liver Disorders* (BLD), *Credit Approval* (CRED), *Pima Indians Diabetes* (PID) and *Boston housing* (HOUS).

The data sets were preprocessed and discretized in accordance with the LAD discretization approach in [45]. Following [16], we only used level variables and did not introduce interval variables. We call these preprocessed data sets `BCW_simple`, `HD_simple`, `BLD_simple`, `CRED_simple`, `PID_simple` and `HOUS_simple`.

We compared the running times to compute *all* patterns of a given pattern type of the implementation using `asprin` from Sect. 4.5 with the ones in Sect. 4.3 and 4.4, using hard constraints and `no asprin`. The running times are shown in Table 2, with the number of computed patterns in parentheses. The faster of the two approaches is marked in gray.

The first observation is that there is no clear winner when comparing the two approaches. Only for spanned patterns, the original implementation is faster than `asprin` in all the experiments. For all other pattern types, the relative efficiency of the two approaches depends on the underlying data set. For both methods the running time normally increases when many patterns have to be calculated, which can be explained by the time needed for enumeration. Taking a closer look at the spanned patterns, the `no asprin` implementation seems to benefit from the fact that many patterns that are not optimal are calculated by `asprin`. For the CRED data set, for example, `asprin` generates 2809 models in total, of which only 672 are optimal. The implementation without `asprin` does not use optimization at all and therefore might be faster.

For all pattern types including the evidential preference, i.e., strong, strong prime and strong spanned patterns, the experiments suggest that for smaller data sets (at the bottom of Table 2) our original implementation requires less running time, while for the larger data sets (at the top) the `asprin` approach is more efficient. This behavior may be attributed to the more restricted search space in the implementations using hard constraints. A more constrained search space leads to faster results for smaller instances. However, the `asprin` approach uses significantly fewer rules, which leads to less overhead and rules after grounding. This makes the `asprin` implementation more efficient for larger instances.

5.2 Efficient pattern generation

Next we illustrate the efficiency of our ASP approach for pattern generation on larger problem instances. We use again the six data sets from the UC Irvine Machine Learning Repository [44], but before the feature selection step, which leads to larger problem sizes. We call these data sets `BCW_full`, `HD_full`, `BLD_full`, `CRED_full`, `PID_full` and `HOUS_full`. The resulting problem sizes (rows $\times$ columns) are given in Table 3. For each of the pattern types *prime*, *strong*, *spanned*, *strong prime* and *strong spanned*, we show the CPU time needed for the generation of the first pattern, of the first 100 patterns and the first 1000 patterns. We used a timeout of 6 hours.

All tasks aiming to generate one pattern with the given characteristics resulted in a running time of a few seconds. The increasing time required for the calculation from the first to the last row in Table 3 is due to the size of the data sets. The larger the data set, the more time is needed for *grounding* the problem instances. The time needed for generating the first 100 patterns depends on the pattern type. Here, the problem of strong pattern calculation can be solved most quickly. Indeed, the running times for calculating the first 100 patterns do not differ much from the times needed for generating the first pattern only. This observation is

Table 2 CPU times for the generation of *all* patterns of the different pattern types with and without asprin

	Prime		Strong		Spanned		Strong prime		Strong spanned	
	asprin	no asprin	asprin	no asprin	asprin	no asprin	asprin	no asprin	asprin	no asprin
BCW_simple	13.9s (95)	10.7s (95)	1.7s (8407)	9.3s (8407)	4d14h06m0.3s (37758)	6.4s (37758)	7.4s (27)	13.3s (27)	2.2s (26)	1m10.6s (26)
HD_simple	2.9s (100)	1.5s (100)	0.7s (145)	1.7s (145)	38.2s (493)	0.4s (493)	2.8s (34)	3.9s (34)	1.4s (29)	2.6s (29)
BLD_simple	0.7s (9)	2.6s (9)	2.3s (28800)	1m10.2s (28800)	1.0s (6)	0.1s (6)	0.5s (4)	24.3s (4)	0.8s (4)	19.9s (4)
CRED_simple	2m44.6s (324)	35.7s (324)	2.8s (37656)	29.6s (37656)	5m58.0s (672)	0.5s (672)	20.7s (68)	38.2s (68)	3.6s (32)	32.6s (32)
PID_simple	41.0s (131)	5.2s (131)	44.8s (955400)	38.9s (955400)	14.1s (89)	0.1s (89)	0.9s (7)	1.9s (7)	1.0s (6)	0.4s (6)
HOUS_simple	6.7s (70)	0.9s (70)	0.5s (1120)	0.1s (1120)	2.7s (27)	0.0s (27)	0.5s (6)	0.3s (6)	0.3s (2)	0.0s (2)

Simplified data sets from [44], after discretization and feature selection. The number of calculated patterns is shown in parentheses

Table 3 Running times (CPU) for pattern generation of different types on six data sets from the UC Irvine Machine Learning Repository [44]

	size	prime first	prime first 100	prime first 1000
BCW_full	449 × 72	1.6s	9m54s	(≥ 6 hours)
HD_full	297 × 305	3.3s	11.4s	12m52s
BLD_full	341 × 269	3.3s	11.7s	9m27s
CRED_full	653 × 773	17.9s	42.4s	20m59s
HOUS_full	506 × 1217	25.6s	1m57s	36m49s
PID_full	768 × 857	24.6s	51.1s	22m57s

	strong first	strong first 100	strong first 1000	spanned first	spanned first 100	spanned first 1000
BCW_full	1.6s	1.3s	2.4s	1.7s	8m04s	244m24s
HD_full	3.3s	3.3s	5.6s	4.1s	2m04	161m01s
BLD_full	3.2s	3.4s	4.6s	4.9s	4m32s	117m56s
CRED_full	17.6s	18.8s	26.2s	22.8s	6m59s	337m21s
HOUS_full	21.8s	23.4s	41.1s	41.3s	19m15s	(≥ 6 hours)
PID_full	23.2s	24.7s	40.9s	29.5s	13m26s	(≥ 6 hours)

	strong prime first	strong prime first 100	strong prime first 1000	strong spanned first	strong spanned first 100	strong spanned first 1000
BCW_full	1.5s	11m04s	(≥ 6 hours)	1.8s	6m01s	121m01s
HD_full	3.7s	2m26s	66m21s	4.8s	42m29s	(≥ 6 hours)
BLD_full	3.8s	57.1s	49m01s	6.9s	28m59s	(≥ 6 hours)
CRED_full	22.3s	2m29s	94m20s	31.6s	(≥ 6 hours)	(≥ 6 hours)
HOUS_full	35.2s	7m22s	123m33s	58.2s	(≥ 6 hours)	(≥ 6 hours)
PID_full	27.9s	1m50s	62m17s	42.4s	(≥ 6 hours)	(≥ 6 hours)

confirmed by the times for calculating the first 1000 strong patterns. All problems could be solved in less than 1 minute.

The calculation of prime patterns is much more time consuming. We see in Table 3 that calculating the first 100 patterns requires several seconds up to 10 minutes, while calculating 1000 patterns takes between 10 to 37 minutes. In the case of the BCW data set, the calculation was not completed within the 6-hour time limit. Even more difficult is the generation of spanned patterns. The running times for the first 100 spanned patterns lie in a range from 2 to 20 minutes. The calculation of 1000 patterns takes about 2 to 5 hours for the data sets BCW, HD, BLD and CRED. The problem was not solved for the HOUS and the PID data set within the timeout.

Our efforts to calculate *all* patterns of the five pattern types for all of the given data sets did not terminate within a time limit of 40 hours, due to the fact that their number is huge. To illustrate, for the BCW data set, 62627 prime patterns, 938965219 strong patterns and 36521 spanned patterns were found before timeout.

In summary, our ASP approach to pattern generation enables us to calculate all common types of LAD patterns on data sets containing several hundred observations and features. The running times grow with the number of patterns we are looking for. Since the number of patterns is huge in general, it is often not possible to enumerate *all* patterns in an efficient way. From the practical point of view, this may also be unnecessary, as a solution space with millions of patterns immediately raises the question of how to process this information. In

Sect. 6 we will demonstrate how the flexibility of the ASP framework allows us to address this challenge in a declarative and user-friendly manner.

5.3 Comparison with mixed-integer linear programming

Our ASP approach is also a promising alternative to the commonly used Mixed-Integer Linear Programs (MILP) for pattern generation. To illustrate this, we present experiments on the six data sets BCW, HD, BLD, HOUS, CRED and PID [44] both in the simple and the full version.

5.3.1 Setup

For the comparison, we use the MILP for the generation of maximal patterns proposed by Ryoo et al. [16], which is shown in Fig. 9. The main idea is to minimize the number of observations in the data set that are *not* covered by the generated pattern. This ensures that the calculated pattern is, by definition, a maximal pattern. In order to bring the mathematical formulation in Fig. 9 to a format that is accessible to a MILP solver, we applied the algebraic modeling language ZIMPL [43, 46].

To compare the run time efficiency of our ASP approach and the MILP for generating maximal patterns proposed in [16], we used again the six data sets from the UC Irvine Machine Learning Repository [44]. We considered both the full binary data sets consisting of all the level variables and the simplified data sets including only the selected features after the greedy selection process.

$$
\begin{aligned}
\min_{z,y,d} \quad & \sum_{i\in\Omega^{\bullet}} y_i \\
\text{s.t.} \quad & \sum_{j=1}^{2n} a_{ij} z_j + n y_i \geq d, \quad i \in \Omega^{\bullet} \\
& \sum_{j=1}^{2n} a_{ij} z_j \leq d - 1, \quad i \in \Omega^{\bar{\bullet}} \\
& z_j + z_{n+j} \leq 1, \quad j \in N \\
& \sum_{j=1}^{2n} z_j = d, \\
& \sum_{i\in\Omega^{\bullet}} y_i \leq |\Omega^{\bullet}| - 1, \\
& z \in \{0,1\}^{2n}, \quad y \in \{0,1\}^{|\Omega^{\bullet}|}, \quad d \in \{1,\ldots,n\}.
\end{aligned}
$$

Fig. 9 MILP adapted from [16] for the generation of maximal patterns. Here $\bullet$ stands either for $+$ or $-$ and $\bar{\bullet}$ denotes the opposite sign. The idea is to compute a pattern P that minimizes the number of observations that are not covered by P. The 0-1 variable z_j (resp. z_{n+j}, $j = 1, \ldots, n$) indicates whether literal x_j (resp. $\bar{x}_j$) is present in P. The 0-1 variable y_i describes whether the i-th observation is not covered by P and d denotes the degree of P. The binary numbers a_{ij} (resp. $a_{i,n+j}$) specify the value of the j-th attribute in the i-th observation (resp. its negation)

In [16] computational results for these data sets are presented, after some feature selection. However, the authors do not specify the parameter settings in the greedy selection procedure. Therefore, it was not possible for us to reconstruct exactly the same 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 [16].

To provide also results on data sets that are easily reproducible, we repeated our calculations on the full binary data sets, including all level variables before feature selection. The size of these data sets is comparatively large, but does not depend on any parameters. Moreover, these larger data sets lead to longer running times for both the MILP and the ASP approach, thereby demonstrating the significant difference between the two approaches.

5.3.2 Results

The computational results are given in Tables 4 and 5. For each of the six data sets, we report on the CPU time to generate the first positive maximal pattern and the CPU time to generate all positive maximal patterns, both for the MILP and the ASP approach. This was done first for the simplified data sets with selected features (Table 4), then for the larger sets including all level variables (Table 5). Note that the first positive maximal pattern generated might differ between MILP and ASP. However, the results for all positive maximal patterns are the same in both cases.

While the ASP solver `clingo` provides the option `-opt-mode=optN` to enumerate all optimal solutions, no direct command for the enumeration of all optimal solutions is available in `SCIP`. For the calculation of all positive maximal patterns with `SCIP`, we followed the instructions in the `SCIP` documentation. First, the problem is solved to optimality. Second, a constraint is added to the MILP stating that the objective function value has to be equal to the optimum value calculated before. Finally, the predefined counting option in `SCIP` can be used to collect all feasible solutions for the adjusted MILP. As a consequence, the time reported in Tables 4 and 5 is the sum of the time to find an optimal solution and the time to collect all feasible solutions for the adjusted MILP.

Table 4 includes the results for the small data sets with selected features. All running times for ASP are in the millisecond range and they become longer as more observations are included in the data sets. Note that the number of attributes is in the same range for all data sets in the feature-selected case. In all cases, the running times of the MILP exceed those of the ASP program. However, this set of small benchmarks may not be representative for the overall performance.

Table 4 Running times (CPU) of the MILP and ASP approach for the calculation of positive maximal patterns on the six discretized data sets from the UCI Machine Learning Repository [44] after feature selection

	Discretized data sets after feature selections				
		MILP (solved by `SCIP`)		ASP (solved by `clingo`)	
	size	first	all	first	all
`BCW_simple`	161 × 20	2.41s	4.23s	0.04s	0.03s
`HD_simple`	73 × 11	0.58s	1.34s	0.03s	0.02s
`BLD_simple`	341 × 15	0.05s	3.38s	0.03s	0.02s
`CRED_simple`	114 × 19	0.44s	1.28s	0.02s	0.02s
`PID_simple`	28 × 24	0.78s	0.91s	0.0s	0.02s
`HOUS_simple`	14 × 20	0.03s	0.22s	0.02s	0.02s

Table 5 Running times (CPU) of the MILP and ASP approach for the calculation of positive maximal patterns on the six discretized data sets from the UCI Machine Learning Repository [44] before feature selection

	Discretized original data sets (all level variables included)				
		MILP (solved by SCIP)		ASP (solved by clingo)	
	size	first	all	first	all
BCW_full	449 × 72	11m04s	38m43s	0.14s	0.17s
HD_full	297 × 305	1h13m	(≥ 72 hours)	1.95s	40m22s
BLD_full	341 × 269	46m45s	(≥ 72 hours)	1.09s	(≥ 72 hours)
CRED_full	653 × 773	(≥ 72 hours)	(≥ 72 hours)	7m15s	(≥ 72 hours)
PID_full	768 × 857	(≥ 72 hours)	(≥ 72 hours)	8h07m	(≥ 72 hours)
HOUS_full	506 × 1217	12h04m	(≥ 72 hours)	20.72s	(≥ 72 hours)

Therefore, we evaluate in Table 5 the running times for the larger data sets. These results clearly show the benefits of ASP. The number of observations in Table 5 lies between 297 and 768, and the number of attributes goes up to 1217. The ASP approach 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 7 minutes for the easiest problem BCW and around 5 to 8 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 may be due to the nature of the data itself, but may also indicate that the level of difficulty depends more on the number of observations than on the number of attributes. In fact, the HOUS data set is the only one of the larger data sets that the MILP was able to solve before the timeout. The MILP took around 15 hours, while the ASP approach returned the result in less than 1 minute. For our calculations we used a timeout 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 limit. ASP 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 indicating the time to calculate all maximal patterns, we see that both the MILP and ASP approach reach their limit. The MILP gave a result only for the smallest data set BCW in around 34 minutes and reached the timeout for all other tasks. ASP needed less than a second to produce the result for the BCW dataset. ASP also found all maximal patterns for the HD data set in approximately 2 hours.

This empirical study clearly shows the benefit of using ASP for pattern generation in LAD. Our ASP program was faster than the MILP in all cases of finding one or all maximal positive patterns. In some of cases, the MILP did not find an answer at all within our time limit of 72 hours, while ASP could still finish the task. The results of this comparison underline the interest of the ASP approach for pattern generation in LAD.

6 Flexibility in pattern generation

In the previous section, we have shown how efficient generation of different pattern types is possible using our ASP approach. However, we also noticed that enumerating *all* patterns for a given data set may not be possible in practice, due to the large number of existing patterns. Here we show how ASP, as a declarative programming paradigm, can be used very naturally to narrow down the solution space in order to generate only those patterns that satisfy the

specific requirements of a given application. Since all pattern characteristics are modeled within ASP, we can easily adapt the pattern generation algorithm to our needs, by changing or adding certain logical rules to the program.

We illustrate this feature of ASP by three characteristics of patterns, each of which can be implemented by adding just one line to our programs. Of course, many more characteristics could be realized in a similar way.

A first example is the degree of the patterns generated. To compute the set of prime patterns of a specific degree d, we add d to the choice rule at the beginning of the program to fix the number of literals chosen for each pattern. Thus, to generate only the patterns of degree 2 we write

```
2{pat(S,B):i(sign,_,S,B)}2.
```

In the example of the BLD data set, this will result in 3315 many prime patterns of degree 2 calculated in about 1.5 hours.

A second requirement that can be easily incorporated is the occurrence of specific literals in the solution patterns. To implement this in ASP, we add an integrity constraint to the program. For example, to ensure that literal x_3 is included in each of the solution patterns, we simply write:

```
:- not pat(3,1).
```

For the BLD data set, this additional constraint narrows down the solution space to five degree 2 prime patterns, namely

$$x_3 \wedge \overline{x_{183}}, \quad x_3 \wedge x_{91}, \quad x_3 \wedge \overline{x_{147}}, \quad x_3 \wedge \overline{x_{23}}, \quad x_3 \wedge x_{269}.$$

The coverage of a pattern is an important aspect in almost any application. Our programs calculate the set of covered observations of a solution pattern to ensure that the generated set of literals meets the definition of a pattern. This allows us to output with each answer set the resulting coverage, using the command `#show covered/2` at the end of the program. By adding an integrity constraint for the coverage, we can restrict the set of solutions to those patterns covering some given observations. For example, by adding

```
:- not covered(1,105).
```

at the end of the program, we ensure that the positive observation 105 is covered by the pattern generated. Running the program on the BLD data set results in a single prime pattern of degree 2 including literal x_3 and covering observation 105:

$$x_3 \wedge \overline{x_{23}}$$

This pattern also covers observation 22 and 312 of the data set, which can be seen in the `clingo` output.

In conclusion, the ASP environment and its concise yet comprehensive modeling language facilitate the adaptation of programs for pattern generation to specific requirements, thus enabling the computation of targeted solutions.

7 Conclusion

In this article, we presented ASP programs for all commonly used pattern types in LAD. The nature of the problems, namely enumeration of patterns, optimization of preferences and handling of Boolean functions makes ASP a perfect environment for these tasks.

While several software tools for LAD pattern generation have been developed in the past [47–49], they are mostly no longer maintained and difficult to extend. The ASP programs for pattern generation that we presented are concise and succinct, which is characteristic for ASP. We are aware that ASP may not yet be as widely used as MILP and future developers might initially have to familiarize themselves with its syntax and semantics. However, we believe that our ASP programs are not only easy to use, but also understandable and can be extended with little effort. To illustrate this, we gave several ideas in Sect. 6 of how our programs can be adapted to specific requirements of the user.

In addition to offering the only software tool for LAD pattern generation that, to our knowledge, includes so many different pattern types, we have shown in Sect. 5.2 that our programs are time-efficient for data sets with up to several hundred observations and features. For the class of maximal patterns, we illustrated how our ASP program outperforms the MILP approach in [16].

In conclusion, we believe that our ASP pattern generation programs are of considerable value for LAD research, as well as applications in a range of diverse fields.

Supplementary information

Software and data sets are available at https://github.com/katinkab/AnswerSetLAD.

Acknowledgements We thank Prof. Torsten Schaub and Prof. Martin Gebser for sharing their expertise on Answer Set Programming.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Crama, Y., Hammer, P.L., Ibaraki, T.: Cause-effect relationships and partially defined boolean functions. Ann. Oper. Res. **16**(1), 299–325 (1988). https://doi.org/10.1007/BF02283750
2. Alexe, G., Alexe, S., Bonates, T.O., Kogan, A.: Logical analysis of data - the vision of Peter L. Hammer. Ann. Math. Artif. Intell. **49**(1–4), 265–312 (2007). https://doi.org/10.1007/S10472-007-9065-2
3. Lejeune, M., Lozin, V., Lozina, I., Ragab, A., Yacout, S.: Recent advances in the theory and practice of logical analysis of data. Eur. J. Oper. Res. **275**(1), 1–15 (2019). https://doi.org/10.1016/j.ejor.2018.06.011
4. Bruni, R., Bianchi, G., Dolente, C., Leporelli, C.: Logical analysis of data as a tool for the analysis of probabilistic discrete choice behavior. Comput. Oper. Res. (2018). https://doi.org/10.1016/j.cor.2018.04.014
5. Hammer, P.L., Bonates, T.O.: Logical analysis of data - an overview: From combinatorial optimization to medical applications. Ann. Oper. Res. **148**(1), 203–225 (2006). https://doi.org/10.1007/S10479-006-0075-Y
6. Alexe, G., Alexe, S., Axelrod, D.E., Hammer, P.L., Weissmann, D.: Logical analysis of diffuse large b-cell lymphomas. Artif. Intell. Med. **34**, 235–67 (2005). https://doi.org/10.1016/j.artmed.2004.11.004

7. Alexe, G., Alexe, S., Liotta, L.A., Petricoin, E., Reiss, M., Hammer, P.L.: Ovarian cancer detection by logical analysis of proteomic data. Proteomics **4**(3), 766–83 (2004). https://doi.org/10.1002/pmic.200300574
8. Lauer, M., Alexe, S., Pothier, C., Blackstone, E., Ishwaran, H., Hammer, P.L.: Use of the logical analysis of data method for assessing long-term mortality risk after exercise electrocardiography. Circulation **106**, 685–90 (2002). https://doi.org/10.1161/01.CIR.0000024410.15081.FD
9. Mortada, M., Yacout, S., Lakis, A.: Fault diagnosis in power transformers using multi-class logical analysis of data. J. Intell. Manuf. **25**(6), 1429–1439 (2014). https://doi.org/10.1007/S10845-013-0750-1
10. Bruni, R., Bianchi, G., Dolente, C., Leporelli, C.: Logical analysis of data as a tool for the analysis of probabilistic discrete choice behavior. Comput. Oper. Res. **106**, 191–201 (2019). https://doi.org/10.1016/J.COR.2018.04.014
11. Das, T.K., Adepu, S., Zhou, J.: Anomaly detection in industrial control systems using logical analysis of data. Comput. Secur. **96**, 101935 (2020). https://doi.org/10.1016/J.COSE.2020.101935
12. Osman, H.: Cost-sensitive learning using logical analysis of data. Knowl. Inf. Syst. **66**(6), 3571–3606 (2024). https://doi.org/10.1007/S10115-024-02070-1
13. James, G., Witten, D., Hastie, T., Tibshirani, R., Taylor, J.: An Introduction to Statistical Learning. Springer, Switzerland (2023). https://doi.org/10.1007/978-3-031-38747-0
14. Boros, E., Crama, Y., Hammer, P., Ibaraki, T., Kogan, A., Makino, K.: Logical analysis of data: classification with justification. Ann. Oper. Res. **188**, 33–61 (2011). https://doi.org/10.1007/s10479-011-0916-1
15. Hammer, P.L., Kogan, A., Simeone, B., Szedmák, S.: Pareto-optimal patterns in logical analysis of data. Discrete Appl. Math. **144**(1), 79–102 (2004). https://doi.org/10.1016/j.dam.2003.08.013
16. Ryoo, H.S., Jang, I.-Y.: MILP approach to pattern generation in logical analysis of data. Discrete Appl. Math. **157**(4), 749–761 (2009). https://doi.org/10.1016/j.dam.2008.07.005
17. Chou, C., Bonates, T.O., Lee, C., Chaovalitwongse, W.A.: Multi-pattern generation framework for logical analysis of data. Ann. Oper. Res. **249**(1–2), 329–349 (2017). https://doi.org/10.1007/S10479-015-1867-8
18. Yan, K., Ryoo, H.S.: 0-1 multilinear programming as a unifying theory for LAD pattern generation. Discrete Appl. Math. **218**, 21–39 (2017). https://doi.org/10.1016/j.dam.2016.08.007
19. Yan, K., Ryoo, H.S.: A multi-term, polyhedral relaxation of a 0–1 multilinear function for Boolean logical pattern generation. J. Glob. Optim. **74**(4), 705–735 (2019). https://doi.org/10.1007/s10898-018-0680-8
20. Ouyang, R., Chou, C.: Integrated optimization model and algorithm for pattern generation and selection in logical analysis of data. Comput. Oper. Res. **124**, 105049 (2020). https://doi.org/10.1016/J.COR.2020.105049
21. Lancia, G., Serafini, P.: Computational complexity and ILP models for pattern problems in the logical analysis of data. Algorithms **14**(8), 235 (2021). https://doi.org/10.3390/A14080235
22. Yan, K., Ryoo, H.S.: Graph, clique and facet of Boolean logical polytope. J. Glob. Optim. **82**(4), 1015–1052 (2022). https://doi.org/10.1007/s10898-021-01107-x
23. Lifschitz, V.: What is answer set programming? In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, pp. 1594–1597 (2008). http://www.aaai.org/Library/AAAI/2008/aaai08-270.php
24. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, USA (2003). https://doi.org/10.1017/CBO9780511543357
25. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press, USA (2014). https://doi.org/10.1017/CBO9781139342124
26. Lifschitz, V.: Answer Set Programming. Springer, Switzerland (2019). https://doi.org/10.1007/978-3-030-24658-7
27. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer, Switzerland (2022). https://doi.org/10.1007/978-3-031-01561-8. Reprint of original edition Morgan & Claypool 2013
28. Becker, K.: Logical analysis of biological data. PhD thesis, Freie Universität Berlin (2021). https://doi.org/10.17169/refubium-31339
29. Becker, K., Gebser, M., Schaub, T., Bockmayr, A.: Answer set programming for logical analysis of data. In: Proceedings of the 12th International Workshop on Constraint-Based Methods for Bioinformatics, WCB'16, pp. 15–26 (2016). http://cp2016.a4cp.org/program/workshops/wcb2016-proceedings.pdf
30. Alexe, G., Hammer, P.L.: Spanned patterns for the logical analysis of data. Discrete Appl. Math. **154**(7), 1039–1049 (2006). https://doi.org/10.1016/j.dam.2005.03.031
31. Alexe, G., Alexe, S., Hammer, P.L.: Pattern-based clustering and attribute analysis. Soft Comput. **10**(5), 442–452 (2006). https://doi.org/10.1007/s00500-005-0505-9

32. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A system for answer set programming. CoRR. cs.AI/0003033 (2000). https://doi.org/10.48550/arXiv.cs/0003033
33. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3), 499–562 (2006). https://doi.org/10.1145/1149114.1149117
34. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Commun. **24**(2), 107–124 (2011). https://doi.org/10.3233/AIC-2011-0491
35. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. Theory Pract. Log. Program **15**(4–5), 449–463 (2015). https://doi.org/10.1017/S1471068415000150
36. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, pp. 1070–1080 (1988)
37. Lifschitz, V.: Twelve definitions of a stable model. In: Banda, M.G., Pontelli, E. (eds.) Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, Proceedings, pp. 37–51 (2008). https://doi.org/10.1007/978-3-540-89982-2_8
38. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1–2), 181–234 (2002). https://doi.org/10.1016/S0004-3702(02)00187-X
39. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., Wanko, P.: Potassco user guide (2019). https://github.com/potassco/guide
40. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: Asprin: customizing answer set preferences without a headache, 1467–1474 (2015). https://doi.org/10.1609/AAAI.V29I1.9398
41. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: A general framework for preferences in answer set programming. Artif. Intell. **325**, 104023 (2023). https://doi.org/10.1016/J.ARTINT.2023.104023
42. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009). https://doi.org/10.1007/S12532-008-0001-1
43. Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., Hulst, R., Kamp, D., Koch, T., Kofler, K., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., Xu, L.: The SCIP Optimization Suite 9.0. ZIB-Report 24-02-29, Zuse Institute Berlin (2024). https://nbn-resolving.org/urn:nbn:de:0297-zib-95528
44. Kelly, M., Longjohn, R., Nottingham, K.: The UCI Machine Learning Repository. https://archive.ics.uci.edu/ml
45. Boros, E., Hammer, P.L., Ibaraki, T., Kogan, A., Mayoraz, E., Muchnik, I.: An implementation of logical analysis of data. Knowl. Data Eng. IEEE Trans. **12**, 292–306 (2000). https://doi.org/10.1109/69.842268
46. Koch, T.: Rapid mathematical prototyping. PhD thesis, Technische Universität Berlin (2004). https://doi.org/10.14279/depositonce-975
47. Mayoraz, E.: C++ tools for logical analysis of data (1998). http://rutcor.rutgers.edu/pub/LAD/man.pdf
48. Bonates, T.O., Gomes, V.S.D.: LAD-WEKA (2014). https://lia.ufc.br/~tiberius/lad/
49. Lemaire, P.: ladoscope: tools for Logical Analysis of Data (2018). http://pit.kamick.free.fr/lemaire/software-lad.html

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.