Knowledge Representation Methods for Bridging Machine Learning and Logical Reasoning

by

Akihiro TAKEMURA

Dissertation

submitted to the Department of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
September 2022

Abstract

The integration of high-level symbolic reasoning with efficient low-level perception is one of the longstanding challenges in the field of artificial intelligence. This thesis explores various approaches and techniques for the integration of machine learning and knowledge representation.

More specifically, in one of the chapters, we shall study how a symbolic method of programming can be used to explain the behavior and predictions of decision tree-ensemble models. In another chapter, we shall present a method to embed logic programs with non-monotonic semantics into matrices, then using a differentiable method to find vectors that correspond to valid interpretations of the program in continuous vector spaces.

The main contributions of this thesis are: (1) presenting a novel application of Answer Set Programming for explaining trained machine learning models, where we use ASP to generate explainable rule sets from tree-ensemble models, and (2) developing a method for computing supported models of normal logic programs in vector spaces using differentiable computation, where normal logic programs are embedded into matrices and model search is carried out using gradient information.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, we have seen an increasing number of *artificial intelligence* applications ranging from personal assistants on smartphones, fraud detection in financial services to autonomous driving. Many of these applications are supported by *machine learning* methods that can learn patterns from large amount of data to make predictions without human intervention. The rise in computational power brought by multicore CPU and GPGPU has also contributed to the accelerating trend, and it is taken for granted that learning algorithms take advantage of parallel processing for faster training. With this rapid adoption of AI applications in society, we started seeing issues with purely data-dependent approaches, for example, bias[1], ethics[2] and explainability[3] in AI.

We conjecture that experience from *knowledge representation and reasoning* (*KR&R* or *KR* for short) can be useful in solving some of the issues we face today in AI. KR is a subfield of artificial intelligence, dedicated to the representation of knowledge in a computer understandable format, so that computers can solve complex reasoning problems automatically. Typical characteristics of KR include the use of symbolic means to represent knowledge and formal methods like logic to solving problems, as in

logic programming. Main advantages of symbolic methods in the context of machine learning include (1) data efficiency, where the algorithms can learn from only a handful of examples, as opposed to requiring many data points, and (2) symbolic representation, where the learned rules and concepts are represented in a more human-readable fashion, as opposed to distributed representation. This raises the question: how can we combine the techniques of KR and ML in a mutually beneficial manner?

A similar point of view has been studied in another subfield of AI, which we now call *neuro-symbolic artificial intelligence*. Different spellings and wordings exist, including *neural-symbolic* and *neurosymbolic*, among others. It is generally accepted that the term *neural* refers to the use of artificial neural networks, and *symbolic* refers to AI approaches that are based on symbolic manipulation, which also refers to, but not limited to, methods based on formal logic. The main interest is then to achieve a best-of-both-worlds scenario, where the strengths of neural and symbolic approaches are combined to benefit each other. Looking from the symbolic side, the strengths of neural approaches are: (1) the ability to learn from "raw" data, such as images and texts, that include noises and imperfections (2) generalization capability, where the trained model can be tested against unseen data. On the other hand, looking from the neural side, the strengths of symbolic approaches are: (1) high explainability, thanks to explicit symbolic representation, (2) the ease with which human expert knowledge can be incorporated into the design, and (3) provable correctness and soundness of reasoning process itself.

The potential benefits of combining machine learning and/or neural approaches with knowledge representation and reasoning approaches have prompted researchers to work in this area. However, there is much to be explored, especially in terms of concrete implementations of these integrated systems. The purpose of this thesis is therefore, in the broadest sense, to explore various approaches and techniques for the integration of ML and KR. More specifically, in one of the chapters, we shall study how a symbolic method of programming (*Answer Set Programming*) can be used to explain the behavior and predictions of decision tree ensembles. In another chapter, we shall present a method to embed logic programs with non-monotonic semantics into matrices, then using a differentiable method to find vectors that correspond to valid interpretations of the program in continuous vector spaces.

1.1 Contributions

The main contributions of this thesis are summarized in this section.

Chapter 3 presents a method for generating rule sets as global and local explanations for tree-ensemble learning methods using Answer Set Programming. To this end, we adopt a decompositional approach where the split structures of the base decision trees are exploited in the construction of rules, which in turn are assessed using pattern mining methods encoded in ASP to extract interesting rules.

The main contributions are:

- We present a novel application of Answer Set Programming (ASP) for explaining trained machine learning models. We propose a method to generate explainable rule sets from tree-ensemble models with ASP. More broadly, this work contributes to the growing body of knowledge on integrating symbolic reasoning with machine learning.
- We present how the rule set generation problem can be reformulated as an optimization problem, where we leverage existing knowledge on declarative pattern mining with ASP.
- We show how both global and local explanations can be generated by our approach, while comparative methods tend to focus on either one exclusively.
- To demonstrate the practical applicability of our approach, we provide both qualitative and quantitative results from evaluations with public datasets, where machine learning methods are used in a realistic setting.

For reproducibility, we have made the code, datasets and experimental settings publicly available in the following GitHub repository: https://github.com/atakemura/treetap

Potions of this chapter were previously published as:

1. A. Takemura and K. Inoue, "Generating Explainable Rule Sets from Tree-Ensemble Learning Methods by Answer Set Programming," in *Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (Virtual Event), 20-27th September 2021* (A. Formisano,

Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens, and N.-F. Zhou, eds.), vol. 345 of *EPTCS*, pp. 127–140, 2021

2. A. Takemura and K. Inoue, "Rule Extraction from Decision Tree Ensembles by Answer Set Programming," Sept. 2020. Poster presentation at 17th International Conference on Principles of Knowledge Representation and Reasoning (KR)

Chapter 4 presents a method for computing supported models of normal logic programs in vector spaces using gradient information. First, the program is translated into a definite program and embedded into a matrix representing the program. We introduce a loss function based on the implementation of the immediate consequence operator T_P by matrix-vector multiplication with a suitable thresholding function, and we incorporate regularization terms into the loss function to avoid undesirable results. We report the results of several experiments where our method shows promising performance when used with adaptive gradient update.

The main contributions are:

- Presenting an alternative method for embedding logic programs into matrices, and designing an almost everywhere differentiable thresholding function.
- Introducing a loss function with regularization terms for computing supported models, and integrating various gradient update strategies.
- Demonstrating with a help of systematic performance evaluation on a range of programs, that by selecting appropriate components, it is possible to achieve much higher performance and stability than the existing method.

Potions of this chapter were previously published as:

1. A. Takemura and K. Inoue, "Gradient-Based Supported Model Computation in Vector Spaces," in *(to appear) 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2022)*, 2022

2. A. Takemura and K. Inoue, "Gradient-Based Supported Model Computation in Vector Spaces," in *Proceedings of the International Conference on Logic Programming 2021 Workshops Co-Located with the 37th International Conference on Logic Programming (ICLP 2021), Porto, Portugal (Virtual), September 20th-21st, 2021*

(J. Arias, F. A. D'Asaro, A. Dyoub, G. Gupta, M. Hecher, E. LeBlanc, R. Peñaloza, E. Salazar, A. Saptawijaya, F. Weitkämper, and J. Zangari, eds.), vol. 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021

For reproducibility, we have made the code, datasets and experimental settings publicly available in the following GitHub repository: https://github.com/atakemura/grasup

2 Background

This chapter provides the necessary background on Logic Programming, Answer Set Programming and Explainability in Machine Learning. Each chapter in this thesis introduces the necessary background, formalization and definitions that are specific to that chapter. Thus, for convenience, we present only the definitions that are common to all of them, and we introduce these topics in a rather informal manner in this chapter.

2.1 Logic Programming

Logic Programming is a declarative form of programming based on formal logic. A logic program typically consists of a set of rules about a particular problem domain. The rules are written in the form:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_m (m \geq 0) \tag{2.1}$$

where h is the head of the rule ($head(r) = h$), and $b_1, ..., b_m$ are *propositional atoms* (*literals*) that constitute the *body* ($body(r) = \{b_1, ..., b_m\}$) of the rule. We use the symbol $\leftarrow$ for *implication*, and $\wedge$ for *logical and*. The rule (2.1) can be read in plain English as "h is true if all of $b_1 \wedge b_2 \wedge \cdots \wedge b_m$ holds".

More formally, we consider a language $\mathcal{L}$ that contains a finite set of propositional variables defined over a finite alphabet and the logical connectives $\neg$, $\wedge$, and $\leftarrow$. The *Herbrand base*, B_P, is the set of all propositional variables in a logic program P. A *definite program* is a set of *rules* of the form (2.1), where h and b_i are propositional variables (*atoms*) in $\mathcal{L}$.

When a rule has no body literals, it is called a $fact$:

$$h \leftarrow \tag{2.2}$$

which states that h is unconditionally true.

When the head of the body is $\bot$, it is called an *integrity constraint*:

$$\bot \leftarrow p \tag{2.3}$$

which states that p cannot be true. $\bot$ in the head is often omitted, thus (2.3) is written as:

$$\leftarrow p \tag{2.4}$$

These are usually used to filter out undesirable situations.

In classical logic, the negation of an atom can be understood as "the atom can be inferred to be false", however, in logic programming, it is often difficult to prove that a statement is false. In logic programming, the notion of *negation as failure* (or *default negation*) is used to denote that, when an atom cannot be shown to be true, it is assumed to be false. For example, a proposition $\neg q$ under this assumption is true, if q cannot be shown to be true.

A program may contain rules with negation as failure literals, and in such cases the program is said to be a *normal program*. More formally, a normal program is a set of rules of the form (2.5) where h and b_i are propositional variables in $\mathcal{L}$.

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \neg b_{l+2} \wedge \cdots \wedge \neg b_m (m \geq l \geq 0) \tag{2.5}$$

We refer to the positive and negative occurrences of atoms in the body as $body^+(r) = \{b_1, \ldots, b_l\}$ and $body^-(r) = \{b_{l+1}, \ldots, b_m\}$, respectively. A normal program is a definite program if $body^-(r) = \emptyset$ for every rule $r \in P$.

2.1.1 Semantics of Logic Programs

An *Herbrand interpretation*, I, of a normal program P is a subset of B_P. A *model* M of P is an interpretation of P where for every rule $r \in P$ of the form (2.5), $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$ imply $h \in M$. A program is called *consistent* if it has a model. A model M of program P is *minimal* if no proper subset of M is a model of P.

The immediate consequence operator $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is a function on Herbrand interpretations. For a definite program P with an interpretation I, we have:

$$T_P(I) = \{head(r) | r \in P \,\text{and}\, body(r) \subseteq I\} \tag{2.6}$$

Essentially, $T_P(I)$ shows what can be deduced from an interpretation I in a single step (hence the name). The powers of T_P are defined as $T_P^{k+1}(I) = T_P(T_P^k(I))$ $(k \geq 0)$ and $T_P^0(I) = I$ [8]. Given $I \subseteq B_P$, there is a fixed-point $T_P^{k+1}(I) = T_P^k(I)$ $(k \geq 0)$. For a definite program P, the least fixed-point $T_P^k(\emptyset)$ coincides with the least model of P [9].

For a normal program, the immediate consequence operator T_P is defined as:

$$T_P(I) = \{head(r) | r \in P \,\text{and}\, body^+(r) \subseteq I \,\text{and}\, body^-(r) \cap I = \emptyset\} \tag{2.7}$$

For normal logic programs in general, unlike the case for definite programs, computing the powers of $T_P(\emptyset)$ is not guaranteed to converge to a fixed-point due to the non-monotonicity.

A *supported model* M is a model of P where for every $p \in M$, there exists a rule $r \in P$ such that $head(r) = p$, $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$ [8, 10]. In other words, in supported models, every atom in the model is "explained" or "covered" by some rules in the program. It is known that a supported model M of a program P is a fixed point of T_P, i.e. $T_P(M) = M$[10].

The stable model semantics [11] defines the semantics of normal logic programs and is the basis of *Answer Set Programming*. Informally, the main idea of the stable model semantics is that, given a set of I atoms from the language of P, we simplify P

by partially evaluating all rules containing the negated versions of I, then checking whether the simplified program P^I has a Least Herbrand Model (LHM). More concretely:

Definition 1 (Gelfond-Lifschitz reduct[11]). *Let P be a ground propositional normal logic program, and I be an Herbrand interpretation. Then, the reduct with respect to I is given by:*

$$P^I = \{head(r) \leftarrow body^+(r) | r \in P, body^-(r) \cap I = \emptyset\} \tag{2.8}$$

By definition, P^I is a definite program. I is a stable model of P if the least Herbrand model of P^I coincides with I.

$$I = LHM(P^I) \tag{2.9}$$

where $LHM(P^I)$ denotes the least Herbrand model of the definite program P^I. The reduct P^I can be obtained procedurally by:

1. delete any rules in P that has a negative literal $\neg b$ in its body where $b \in I$.
2. delete every literal of the form $\neg b$ in the bodies of the remaining rules.

Stable models are also supported models of the program, but the converse does not always hold true. Consider the following program:

$$p \leftarrow q$$
$$q \leftarrow p$$

In this program, there are two supported models, namely $\emptyset$ and $\{p, q\}$, however, only $\emptyset$ is stable.

2.2 Answer Set Programming

Answer Set Programming (*ASP*) [12] has its roots in logic programming and non-monotonic reasoning. It is a declarative programming paradigm often used for solving hard combinatorial and constraint optimization problems. Unlike query-based logic programming languages like Prolog, ASP follows a model generation approach using efficient grounders and solvers (Figure 2.1).

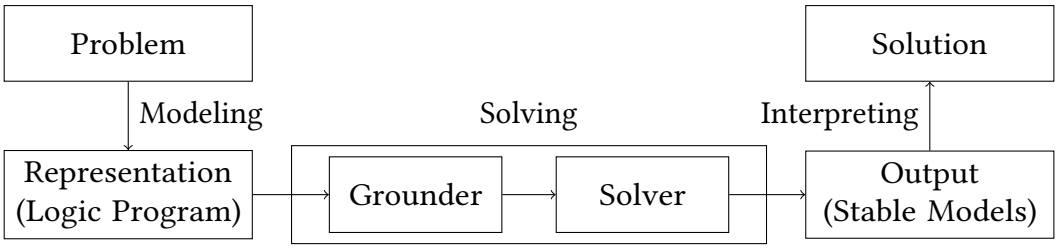


Figure 2.1: A simplified workflow diagram of Answer Set Programming.

The remainder of this section introduces some of the features of modern ASP solvers, however, this section is not meant to be a complete introduction into ASP. ASP is an active and rich research field, and we refer interested readers to Lifschitz ([12, 13]) and Gebser et al. ([14, 15]) for a more thorough introduction. The semantics of ASP (stable model semantics) will be covered in more detail in Chapter 4. We will use the *clingo*[15] system[1] for the remainder of this thesis[2].

In ASP, the logic programs are finite sets of rules of the form (2.5) in the previous section (Section 2.1). A normal logic program induces a collection of intended interpretations, which are called *answer sets* (*stable models*), defined by the stable model semantics [11].

Additionally, in modern ASP systems, constructs such as *conditional literals* and *cardinality constraints* are supported. The former in clingo [14] are written in the form:

$$p \leftarrow q : r. \tag{2.10}$$

which yields p whenever either r is false (thus it does not matter q holds or not), or both q and r are true. The latter are written in the form

$$s_1 \ \{a_1, \ldots, a_m\} \ s_2. \tag{2.11}$$

where s_1 and s_2 are integer constants. The above rule means that "between s_1 and s_2 atoms from $a_1, \ldots, a_m$ are true". Thus, s_1 and s_2 are treated as lower and upper bounds, respectively.

[1] https://potassco.org/clingo/

[2] More specifically, the syntax we show is that of clingo 5.4. The languages of clingo 4 and above are not fully backward compatible.

Aggregation actions like *#count* and *#sum* are also supported:

$$\#sum\ \{\ 2\ :\ apple\ ;\ 3\ :\ banana\ ;\ 5\ :\ orange\ \}\ \leq\ 10. \tag{2.12}$$

This aggregate atom evaluates to true if the sum of the weights inside the curly braces are equal to or less than 10. For example, $\{apple, orange\}$ is a valid answer because the sum condition $((2 + 5) \leq 10)$ holds. The minimization (or maximization) can be expressed with *#minimize* (or *#maximize*) statements.

The aforementioned features are best described with an example. Consider the *Traveling Salesperson Problem*, where the goal is to find the shortest possible route, visiting each city once and returning to the city of origin, given a set of cities and distances between them. A problem instance may be given as[3]:

```
start(a).
city(a). city(b). city(c). city(d).
road(a,b,10). road(b,c,20). road(c,d,25). road(d,a,40).
road(b,d,30). road(d,c,25). road(c,a,35).
```

Then, this problem can be solved with:

```
{ travel(X,Y) } :- road(X,Y,_).
% visit Y by traveling from X to Y
visited(Y) :- travel(X,Y), start(X).
% Y is visited if there's travel(X,Y) and X is visited
visited(Y) :- travel(X,Y), visited(X).
% you have to visit all cities
:- city(X), not visited(X).
% you cannot travel from X to Y more than once
:- city(X), 2 { travel(X,Y) }.
% you cannot travel from Y to X more than once
:- city(X), 2 { travel(Y,X) }.
% now minimize the distance traveled
#minimize { D,X,Y : travel(X,Y), road(X,Y,D) }.
```

[3]This example was taken from [16]

```
#show travel/2.
```

which gives:

```
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
```

with cost of 95 (only optimum answer is shown here).

Moreover, this example illustrates the declarative nature of ASP: there is no information about exactly "how" to solve the problem in this encoding, instead the user only has to provide information about "what" the problem is. The ASP solver clingo in this instance, (*gringo* system) first removes all function variables by *grounding*, then proceeds to calculate the stable models by an answer set solver (*clasp* system). Thus, the grounded program may look like:

```
:~travel(a,b).[10@0,a,b]
:~travel(b,c).[20@0,b,c]
% ... omitted
:-2<=#count{0,travel(d,a):travel(d,a);0,travel(c,a):travel(c,a)}.
:-2<=#count{0,travel(b,c):travel(b,c);0,travel(d,c):travel(d,c)}.
% ... omitted
visited(b):-travel(a,b).
visited(c):-visited(b),travel(b,c).
visited(d):-visited(b),travel(b,d).
% ... omitted
:-not visited(a).
:-not visited(b).
% ... omitted
#show travel/2.
```

Notice that the minimization statement is translated to weighted weak constraints, and that the integrity constraint on the number of visits to the same city is "expanded" to include all possible undesirable cases.

2.3 Explainability in Machine Learning

The past decade saw a rapid rise in adoption of black-box machine learning models in automated decision systems. This rapid spread is made possible by the large amount of data available to machine learning practitioners; however, the data may contain human biases and prejudices, leading to unfair or wrong decisions. The European Union *General Data Protection Regulation* (*GDPR*) which took effect in 2018, introduced "right to explanation" clauses, allowing individuals to access "meaningful information about the logic involved" when automated decision-making takes place[4]. Although the legal scope of these clauses is still debated, this nonetheless shows the importance of explaining black-box decision systems.

The importance of explanation means that it is relevant in multiple industry sectors and scientific disciplines. In reality, each community addresses the problem from different perspectives, and assigns different meaning to the term *explanation*. As of this writing, in the machine learning community, there is no common mathematical definition of explanation that is accepted by the community, and it is still an active research area spanning multiple subfields like natural language processing and computer vision. Thus, it would appear that a brief literature review is in order.

Firstly, let us clarify the two terms that are often used interchangeably in literature: *interpretability* and *explainability*. *Interpretability* in machine learning is the ability to explain or to present in understandable terms to a human [17]. Another definition of interpretability is: how well a human could understand the decisions in the given context [18]. As noted earlier, explainability is often used interchangeably with interpretability, but some emphasize the ability to produce *post-hoc explanations* for the black-box models [19]. Nowadays, it is often assumed that a trained machine learning model is not understandable for humans, however, this is not necessarily the case; for example, one may use simpler models that are recognized as understandable to humans, such as simple decision trees, rules and linear models [20].

Secondly, we discuss the methods by which explainability can be achieved. The example of simpler models that are understandable to humans in the last paragraph are examples of *intrinsic* explainability, that is to say, explainability is achieved by

[4]Article 13(2)(f): "The existence of automated decision-making, including profiling, referred to in Article 22(1) and (4) and, at least in those cases, meaningful information about the logic involved, as well as the significance and the envisaged consequences of such processing for the data subject."

restricting the complexity of the machine learning models. On the other hand, one can take an already trained and opaque machine learning models, then try to "reverse engineer" the models by other means which are considered to be explainable. This latter approach is called *post-hoc explainability* (or post-hoc interpretability). As the name implies, post-hoc explanations require a standalone explanation method for reverse engineering, which may be (but not necessarily) a simpler model. Earlier works that popularized the post-hoc explainability approach, LIME [21] and SHAP [22, 23] in particular, used linear models for explanations. A post-hoc explanation method can either be *model specific* or *model agnostic*: a method that only work for certain types of models is model specific, whereas a method that can work with any machine learning model is model agnostic.

Thirdly, explanations can be categorized into either *global explanations* or *local explanations*. Global explanation refers to descriptions of how the overall system works (also referred to as *model explanations*), and local explanation refers to specific descriptions of why a certain decision was made (*outcome explanation*) [20]. The global explanations are more useful in situations where the explanations behind the opaque model is needed, for example, when designing systems for faster detection of certain events such as credit issues or illnesses. In contrast, the local explanations are suitable, for example, when explaining the outcome of such systems to its users, since they are more likely to be interested in particular decisions that led to the outcome.

3 Explaining ML models with ASP

3.1 Introduction

Interpretability in machine learning is the ability to explain or to present in understandable terms to a human [17]. Interpretability is particularly important when, for example, the goal of the user is to gain knowledge from some form of explanations about the data or process through machine learning models, or when making high-stakes decisions based on the outputs from the machine learning models where the user has to be able to trust the models. *Explainability* is another term that is often used interchangeably with interpretability, but some emphasize the ability to produce *post-hoc explanations* for the black-box models [19]. For convenience, we shall use the term *explanation* when referring to post-hoc explanations in this paper.

In this work[1], we address the problem of explaining trained tree-ensemble models

[1]Some parts of this paper were presented as a Technical Communications paper [4] at the 37th International Conference on Logic Programming (ICLP 2021). The present paper newly describes a method to produce explanations for *each predicted instance* (local explanation), in addition to the

by extracting meaningful rules from them. This problem is of practical relevance in business domains, where the understanding of the behavior of high-performing machine learning models and extraction of knowledge in human-readable form can aid users in the decision-making process. We use *Answer Set Programming (ASP)* [11, 12] to generate rule sets from tree-ensembles. ASP is a declarative programming paradigm for solving difficult search problems. An advantage of using ASP is its expressiveness and extensibility, especially when representing constraints. To our knowledge, ASP has never been used in the context of rule set generation from tree-ensembles, although it has been used in pattern mining [24, 25, 26, 27].

Generating explanations for machine learning models is a challenging task, since it is often necessary to account for multiple competing objectives. For instance, if accuracy is the most important metric, then it is in direct conflict with explainability because accuracy favors specialization while explainability favors generalization. Any explanation method should also strive to imitate the behavior of learned models as to minimize misrepresentation of models, which in turn may result in misinterpretation by the user. While there are many explanation methods available (some are covered in Section 2), we propose to use ASP as a medium to represent the user requirements declaratively and to quickly search feasible solutions for faster prototyping. By implementing a rule selection method as a post-processing step to model training, we aim to offer an off-the-shelf objective explanation tool as an alternative to subjective manual rule selection, which can be applied to existing processes with minimum modification.

To demonstrate the adaptability of our approach, we present implementations for both *global* and *local* explanations of learned tree-ensemble models using our method. In general, *global explanation* refers to descriptions of how the overall system works (also referred to as *model explanation*), and *local explanation* refers to specific descriptions of why a certain decision was made (*outcome explanation*) [20]. The global explanations are more useful in situations where the explanations behind the opaque model is needed, for example, when designing systems for faster detection of certain events such as credit issues or illnesses. In contrast, the local explanations are suitable, for example, when explaining the outcome of such systems to its users, since they are

updated ASP encoding for the global explanation method. The experimental section reports new evaluation results of the updated methods on various datasets, including several additional datasets.

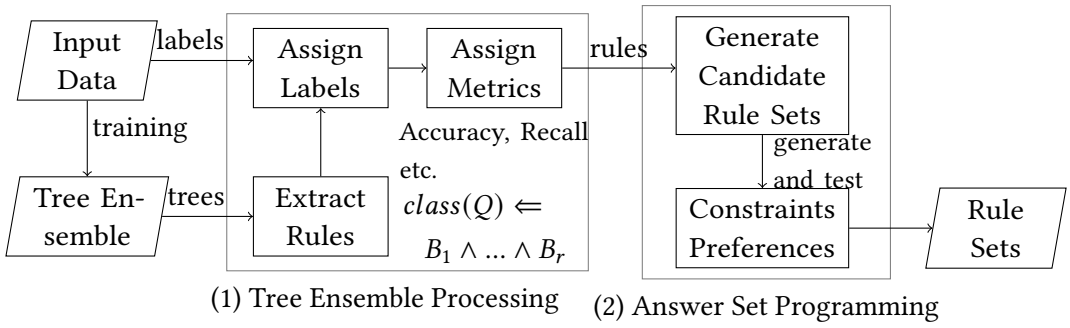


Figure 3.1: Overview of our framework

more likely to be interested in particular decisions that led to the outcome.

We consider the two-step procedure for rule set generation from trained tree-ensemble models (Figure 3.1): (1) extracting rules from tree-ensembles, and (2) computing sets of rules according to selection criteria and preferences encoded declaratively in ASP. For the first step, we employ the efficiency and prediction capability of modern tree-ensemble algorithms in finding useful feature partitions for prediction from data. For the second step, we exploit the expressiveness of ASP in encoding constraints and preference to select useful rules from tree-ensembles, and rule selection is automated through a declarative encoding. In the end, we obtain the generated rule sets therefore as explanations for the tree-ensemble models.

We then evaluate our approach using public datasets. For evaluating global explanations, we use the size and relevance of rules in the rule sets. The number of rules is often associated with explainability, with many rules being less desirable. Additionally, we shall also evaluate the number of conditions in the rule sets, as well as average number of conditions per rule. Performance metrics such as classification accuracy, precision and recall can be used as a measure of relevance of the rules to the prediction task. For evaluating local explanations, we use precision and coverage metrics to compare against existing systems.

This paper makes the following contributions:

- We present a novel application of Answer Set Programming (ASP) for explaining trained machine learning models. We propose a method to generate explainable rule sets from tree-ensemble models with ASP. More broadly, this work contributes to the growing body of knowledge on integrating symbolic reasoning

with machine learning.

- We present how the rule set generation problem can be reformulated as an optimization problem, where we leverage existing knowledge on declarative pattern mining with ASP.
- We show how both global and local explanations can be generated by our approach, while comparative methods tend to focus on either one exclusively.
- To demonstrate the practical applicability of our approach, we provide both qualitative and quantitative results from evaluations with public datasets, where machine learning methods are used in a realistic setting.

The rest of this paper is organized as follows. In Section 2 we review and discuss related works. In Section 3, we review tree-ensembles, ASP and pattern mining. Section 4 presents our method to generate rule sets from tree-ensembles using pattern mining and optimization encoded in ASP. Section 5 describes global and local explanations in the context of our approach. Section 6 presents experimental results on public datasets. Finally, in Section 7 we present the conclusions.

3.2 Related Works

Summarizing tree-ensemble models has been studied in literature, see for example, Born Again Trees [28], defragTrees [29] and inTrees [30]. While exact methods and implementations differ among these examples, a popular approach to tree-ensemble simplification is to create a simplified decision tree model that approximates the behavior of the original tree-ensemble model. Depending on how the approximate tree model is constructed, this could lead to a deeper tree with an increased number of conditions, which makes them difficult to interpret.

Integrating association rule mining and classification is also known, e.g., Class Association Rules (CARs) [31], where association rules discovered by pattern mining algorithms are combined to form a classifier. Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [32] was proposed as an efficient approach for classification based on association rule mining, and it is a well-known rule-based classifier. In

CARs and RIPPER, rules are mined from data with dedicated association rule mining algorithms, then processed to produce a final classifier.

Interpretable classification models is another area of active research. Interpretable Decision Sets (IDS) [33] are learned through an objective function, which simultaneously optimizes accuracy and interpretability of the rules. In Scalable Bayesian Rule Lists (SBRL) [34], probabilistic IF-THEN rule lists are constructed by maximizing the posterior distribution of rule lists. In RuleFit [35], a sparse linear model is trained with rules extracted from tree-ensembles. RuleFit is the closest to our work in this regard, in the sense that both RuleFit and our method extract conditions and rules from tree-ensembles, but differ in the treatment of rules and representation of final rule sets. In RuleFit, rules are accompanied by regression coefficients, and it is left up to the user to further interpret the result.

Lundberg et al. [23] showed how a variant of SHAP [22], which is a post-hoc explanation method, can be applied to tree-ensembles. While our method does not produce importance measures for each feature, the information about which rule fired to reach the prediction can be offered as an explanation in a human-readable format. Shakerin et al. [36] proposed a method to use LIME weights [21] as a part of learning heuristics in inductive learning of default theories. Anchors [37] generates a single high-precision rule as a local explanation with probabilistic guarantees. It should be noted that both LIME and Anchors require the features to be discretized, while recent tree-ensemble learning algorithms can work with continuous features. Furthermore, instead of learning rules with heuristics from data, our method directly handles rules which exist in decision tree models with answer set solver.

Guns et al. applied constraint programming (CP), a declarative approach, to itemset mining [38]. This constraint satisfaction perspective led to the development of ASP encoding of pattern mining [24, 25]. Gebser et al. applied preference handling to sequential pattern mining [26], and Paramonov et al. extended the declarative pattern mining [27] by incorporating dominance programming (DP) from Negrevergne et al. [39] to the specification of constraints. Paramonov et al.[27] proposed a hybrid approach where the solutions are effectively screened first with dedicated algorithms for pattern mining tasks, then declarative ASP encoding is used to extract condensed patterns. While aforementioned works focused on extracting interesting patterns from transaction or sequence data, our focus in this paper is to generate rule sets from

tree-ensemble models to help users interpret the behavior of machine learning models. As for the ASP encoding, we use dominance relations similar to the ones presented in Paramonov et al. [27] to further constrain the search space.

3.3 Background

In the remainder of this paper, we shall use *learning algorithms* to refer to methods used to train *models*, as in machine learning literature. We use *models* and *explanations* to refer to machine learning models and post-hoc explanations about the said models, respectively. For the relevant background material on *Answer Set Programming*, refer to Section 2.2.

3.3.1 Tree-Ensemble Learning Algorithms

Tree-Ensemble (TE) learning algorithms are machine learning methods widely used in practice, typically, when learning from tabular datasets. A trained TE model consist of multiple base decision trees, each trained on an independent subset of the input data. For example, Random Forests [40] and Gradient Boosted Decision Tree (GBDT) [41] are tree-ensemble learning algorithms. Recent surge of efficient and effective GBDT algorithms, e.g., LightGBM [42], has led to wide adoption of TE learning algorithms in practice. Although individual decision trees are considered to be interpretable [43], ensembles of decision trees are seen as less interpretable.

The purpose of using TE learning algorithms is to train models that predict the unknown value of an attribute y in the dataset, referred to as *labels*, using the known values of other attributes $\mathbf{x} = (x_1, x_2, ..., x_m)$, referred to as *features*. For brevity, we restrict our discussion to classification problems. During the training or learning phase, each input instance to the TE learning algorithm is a pair of features and labels, i.e. $(\mathbf{x}_i, y_i)$, where i denotes the instance index, and during the prediction phase, each input instance only include features, $(\mathbf{x}_i)$, and the model is tasked to produce predictions $\hat{y}_i$. A collection of input instances, complete with features and labels, is referred to as a *dataset*. Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ with $n \in \mathbb{N}$ examples and $m \in \mathbb{N}$ features, a decision tree classifier t will predict the class label $\hat{y}_i$ based on the feature vector $\mathbf{x}_i$ of the i-th sample: $\hat{y}_i = t(\mathbf{x}_i)$. A tree-ensemble $\mathcal{T}$ uses $K \in \mathbb{N}$ trees and additionally an

aggregation function f over the K trees which combines the output from the trees: $\hat{y}_i = f(t_{k \in K}(\mathbf{x}_i))$. As for Random Forest, for example, f is a majority voting scheme (i.e. `argmax` of `sum`), and in GBDT f may be a summation followed by softmax to obtain $\hat{y}_i$ in terms of probabilities.

In this paper, a decision tree is assumed to be a binary tree where the internal nodes hold split conditions (e.g., $x_1 \leq 0.5$) and leaf nodes hold information related to class labels, such as the number of supporting data points per class label that have been assigned to the leaf nodes. Richer collections of decision trees provide higher performance and less uncertainty in prediction compared to a single decision tree. Typically, each TE model has specific algorithms for learning base decision trees, adding more trees and combining outputs from the base trees to produce the final prediction. In GBDT, the base trees are trained sequentially by fitting the residual errors from the previous step. Interested readers are referred to [41], and its more recent implementations, LightGBM [42] and XGBoost [44].

3.3.2 Pattern Mining

In a general setting, the goal of pattern mining is to find interesting patterns from data, where patterns can be, for example, itemsets, sequences and graphs. For example, in *frequent itemset mining* [45], the task is to find all subsets of items that occur together more than the threshold count in databases. In this work, the patterns of interest are sets of predictive rules. A *predictive rule* has the form $c \Leftarrow s_1 \wedge s_2 \wedge, ..., s_n$, where c is a class label, and $\{s_i\}$ $(1 \leq i \leq n)$ represents conditions.

For pattern mining with constraints, the notion of *dominance* is important, which intuitively reflects the pairwise preference relation ($<^*$) between patterns [39]. Let C be a constraint function that maps a pattern to $\{\top, \bot\}$, and let p be a pattern, then the pattern p is *valid* iff $C(p) = \top$, otherwise it is *invalid*. An example of C is a function that checks the support of a pattern is above the threshold. The pattern p is said to be *dominated* iff there exists a pattern q such that $p <^* q$ and q is valid under C. Dominance relations have been used in ASP encoding for pattern mining [27].

There are existing ASP encodings of pattern mining algorithms, e.g., [24, 26, 27], that can be used to mine itemsets and sequences. Here, we develop and apply our encoding on rules to extract interesting rules from tree-ensembles. On the surface, our

problem setting may appear similar to frequent itemset and sequence mining; however, rule set generation is different from these pattern mining problems. We can indeed borrow some ideas from frequent itemset mining for encoding; however, our goal is not to decompose rules (cf. transactions) into individual conditions (cf. items) then constructing rule sets (cf. itemsets) from conditions, but rather to treat each rule in its entirety then combining rules to form rule sets. The body (antecedent) of a rule can also be seen as a sequence, where the conditions are connected by conjunction connective $\wedge$, however, in our case, the ordering of conditions does not matter, thus sequential mining encodings that use slots to represent positional constraints [26] cannot be applied directly to our problem.

3.4 Rule Set Generation

3.4.1 Problem Statement

The rule set generation problem is represented as a tuple $P = \{R, M, C, O\}$, where R is the set of all rules in the tree-ensemble, M is the set of meta-data and properties associated with each rule in R, C is the set of user-defined constraints including preferences, and O is the set of optimization objectives. The goal is to generate a set of rules from R by selection under constraints C and optimization objectives O, where constraints and optimization may refer to the meta-data M. In the following sections, we describe how we construct each R, M, C and O, and finally, how we solve this problem with ASP.

3.4.2 Rule Extraction from Decision Trees

Recall that a tree-ensemble $\mathcal{T}$ is a collection of K decision trees, and we refer to individual trees t_k with subscript k. An example of a decision tree-ensemble is shown in Figure 3.2. A decision tree t_k has N_{t_k} nodes and L_{t_k} leaves. Each node represents a split condition, and there are L_{t_k} paths from the root node to the leaves. For simplicity, we assume only features that have orderable values (continuous features) are present in the dataset in the examples below.[2] The tree on the left in Figure 3.2 has 4 internal

[2]Real datasets may have unorderable categorical values. For example, in the *census* dataset, occupation (Sales, etc.) and education (Bachelors, etc.) are categorical features. Support for categorical feature split

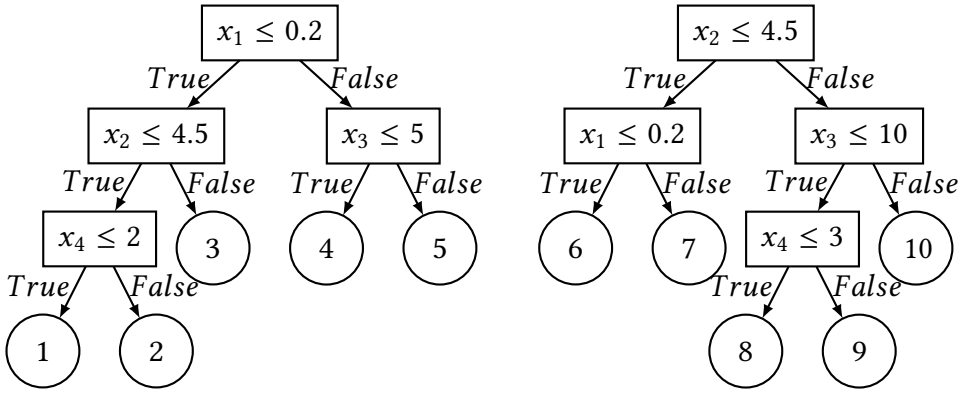


Figure 3.2: A simple decision tree-ensemble consisting of two decision trees. The rule associated with each node is given by the conjunction of all conditions associated with nodes on the paths from the root node to that node.

nodes including the root node with condition $[x_1 \leq 0.2]$ and 5 leaf nodes; therefore there are 5 paths from the root note to the leaf nodes 1 to 5.

From the left-most path of the decision tree on the left in Figure 3.2, the following prediction rule is created. We assume that node 1 predicts class label 1 in this instance.[3]

$$class(1) \Leftarrow (x_1 \leq 0.2) \wedge (x_2 \leq 4.5) \wedge (x_4 \leq 2)$$

Assuming that node 2 predicts class label 0, we also construct the following rule (note the reversal of the condition on x_4):

$$class(0) \Leftarrow (x_1 \leq 0.2) \wedge (x_2 \leq 4.5) \wedge (x_4 > 2)$$

The set of all rules, R, is constructed as follows:

1. Enumerate all possible paths from the root node to the leaves.
2. For each path, at each subsequent node on the path to the leaf node, the split condition of the node is appended to the body (antecedent, set of conditions) of the rule.

is implementation-dependent, however in general one can replace the continuous split with a subset selection e.g., $x_c \in \{x_{c1}, x_{c2}, ...\}$

[3]Label=1 and 0 refer to the attributes in the dataset and have different meaning depending on the dataset. For example, in the *census* dataset, label=1 and 0 mean that the personal income is more than $50,000 and that it is no more than $50,000, respectively.

3. Compute the predicted class label for each rule. For simplicity, we apply all conditions in the rule and calculate the most likely class label from the count data (`argmax` of counts).

4. Add the generated rules to the candidate rule set R.

5. Repeat steps 1 to 4 for each tree t_k where $1 \leq k \leq K$, in the ensemble of K trees.

By constructing the candidate rule set R in this way, the bodies (antecedents) of rules included in rule sets are guaranteed to exist in at least one of the trees in the tree-ensemble. Rule sets generated in this manner are therefore faithful to the representation of the original model in this sense. If we were to construct rules from the unique set of split conditions, the resulting rule may have combinations of conditions that do not exist in any of the trees.

Proposition 1. *If R is constructed according to the steps 1-5, then the bodies of the rules in R exist in at least one of the trees in the tree-ensemble.*

Proof. Suppose there is a rule in R whose body does not exist in any of the trees. Steps 1 enumerates all possible paths from the root to the leaf nodes, and Step 2 follows the paths while constructing the bodies of the rules. A non-existent path is excluded from Step 1. Therefore, if a path does not exist in at least one of the trees in the ensemble, a rule whose body is constructed from a non-existent path cannot be included in R. A contradiction. □

Let us assume that (1) all K trees in the ensemble are perfect binary decision trees and have the same height h, (2) there are n examples and m features in the dataset, and (3) there are no duplicate rules and conditions across trees.

Proposition 2. *The maximum size of R is $K \times (2^{h+1} - 2)$. The reduced size, constructed by only considering the rules at the leaf nodes, is $K \times 2^h$.*

Proof. The reduced size ($K \times 2^h$) follows immediately from the number of leaf nodes in a perfect binary decision tree with height h, i.e., 2^h. The number of internal nodes is $2^h - 1$, and by enumerating possible outcomes (*True* or *False*) of internal nodes we have $2(2^h - 1) = 2^{h+1} - 2$ rules from each tree. □

In practice, there are duplicate split conditions across trees in a tree-ensemble, so the unique count of rules is often smaller than the maximum value.

Proposition 3. *The time complexity of the proposed method to construct R is $O(K \times (2^{h+1} + 2^{h+1} \times n \times h))$ for all rules, and $O(K \times (2^{h+1} + 2^h \times n \times h))$ for the reduced size case.*

Proof. Enumerating paths to leaf nodes takes $O(2^{h+1})$ time by depth first search. There are $O(2^{h+1})$ rules for the all rules case, and $O(2^h)$ rules for the reduced size case. For each rule, all conditions in a rule need to be applied to the data. Since there are at most h conditions in a rule, and there are n examples, it takes $O(n \times h)$ time to apply all conditions in a rule. □

3.4.3 Computing Metrics and Meta-data for Selection

After the candidate rule set R is constructed, we gather information about the performance and properties of each rule and collect them into a set M. The meta-data, or properties, of a rule are information such as the size of the rule, as defined by the number of conditions in the rule, and the ratio of instances which are covered by the rule. Performance metrics measure how well a rule can predict class labels. Here we calculate the following performance metrics: accuracy, precision, recall and F1-score. For classification tasks, a *true positive (TP)* refers to an outcome where the model correctly predicts the positive class, and a *true negative (TN)* refers to an outcome where the model correctly predicts the negative class. Conversely, a *false positive (FP)* refers to an outcome where the model incorrectly predicts the positive class, and a *false negative (FN)* is an outcome where the model incorrectly predicts the negative class.

$$\begin{aligned} accuracy &= \frac{TP + TN}{TP + TN + FP + FN} & precision &= \frac{TP}{TP + FP} \\ recall &= \frac{TP}{TP + FN} & F1\text{-}score &= 2 \times \frac{precision \times recall}{precision + recall} \end{aligned} \tag{3.1}$$

We compute multiple metrics for a single rule, to meet a range of user requirements for explanation. One user may only be interested in simply the most accurate rules (maximize accuracy), whereas another user could be interested in more precise rules (maximize precision), or rules with more balanced performance (maximize F1-score).

Table 3.1: List of predicates representing a rule in ASP.

Predicate	Meaning[a]
`rule(X)`	`X` holds the rule index.
`condition(X,I)`	Rule `X` has condition `I`.
`size(X,L)`	Number of conditions in rule `X` (length, `L`).
`predict_class(X,C)`	Predicted class label `C` of rule `X`.
`support(X,S)`	Support `S` of rule `X`, the ratio of instances that is covered by rule `X`.*
`error_rate(X,E)`	Error rate ($1 - accuracy$), `E`, of the rule `X` evaluated in the training data.
`accuracy(X,A)`	Accuracy score `A` of rule `X`.*
`precision(X,P)`	Precision score `P` of rule `X`.*
`recall(X,R)`	Recall score `R` of rule `X`.*
`f1_score(X,F)`	F1-score `F` of rule `X`.*

[a]Properties and metrics marked with asterisks(*) are multiplied by 100 and rounded to the nearest integer.

The candidate rule set R and meta-data set M are represented as facts in ASP, as shown in Table 3.1. For example, the first rule in Section 3.4.2 may be represented as follows[4]:

```
% rule 1
rule(1). condition(1,1). condition(1,2). condition(1,3).
support(1,10). size(1,3). accuracy(1,50).
error_rate(1,50). precision(1,30).
recall(1,40). f1_score(1,34). predict_class(1,1).
```

3.4.4 Encoding Inclusion Criteria and Constraints

As with previous works in pattern mining in ASP, we follow the "generate-and-test" approach, where a set of potential solutions are generated by a choice rule and subsequently constraints are used to filter out unacceptable candidates. In the context of rule set generation, we use a choice rule to generate candidate rule sets that may constitute a solution. In this section, we introduce the following selection criteria and constraints: (1) individual rule selection criteria that are applied on a per-rule basis, (2)

[4]The performance metrics are for illustration purposes only and are chosen arbitrarily.

pairwise constraints that are applied to pairs of rules, and (3) collective constraints that are applied to a set of rules.

The "generator" choice rule has the following form:

```
% pick at least 1 rule and at maximum 5 rules for each class.
1 { selected(X) :  predict_class(X, K), valid(X) } 5 :- class(K).
```

The choice rule above generates candidate subsets of size between 1 and 5 from R, where we use the `selected/1` predicate to indicate that a rule (`rule(X)`) is included in the subset. Individual rule selection criteria are integrated into the generator choice rule by the `valid/1` predicate, where a rule `rule(X)` is valid whenever `invalid(X)` cannot be inferred.

```
valid(X) :- rule(X), not invalid(X).
```

The following criterion excludes rules with low support from the candidate set:

```
% exclude rules that apply to less than 5% of instances
invalid(X) :- rule(X), support(X,S), S < 5.
```

Pairwise constraints can be used to encode dominance relations between rules. For a rule `X` to be dominated by `Y`, `Y` must be strictly better in one criterion than `X` and at least as good as `X` or better in other criteria. In the following case, we encode the dominance relation between rules using the accuracy metric and support, where we prefer rules that are accurate and covers more data.

```
% cannot be dominated
:- dominated.
% X is dominated by Y
gt_acc_geq_cov(Y) :- selected(X), valid(Y),
    accuracy(X,Ax), accuracy(Y,Ay),
    support(X,Spx), support(Y,Spy),
    Ax < Ay, Spx <= Spy.
geq_acc_gt_cov(Y) :- selected(X), valid(Y),
    accuracy(X,Ax), accuracy(Y,Ay),
    support(X,Spx), support(Y,Spy),
```

```
    Ax <= Ay, Spx < Spy.
dominated :- valid(Y), gt_acc_geq_cov(Y).
dominated :- valid(Y), geq_acc_gt_cov(Y).
```

Collective constraints are applied to collections of rules, as opposed to individual or pairs of rules. The following restricts the maximum number of conditions in rule sets, using the aggregate atom `#sum`:

```
% total number of conditions should not exceed 30
:- #sum { S,X : size(X,S), selected(X) } > 30.
```

We envision two main use-cases for the criteria and constraints introduced in this section: (1) to generate rule sets with certain properties, and (2) to reduce the computation time. For (1), the user can use the individual selection criteria to ensure that the rules included into the candidate rule sets have certain properties, or the collective constraints to put restrictions on the aggregate properties of the rule sets. The latter use-case have more practical relevance because in our case, as in pattern mining, the complexity of a naive "generate-and-test" approach is exponential with respect to the number of candidates.

To reduce the search space, one can place an upper bound on the size of generated candidate sets, and use the `invalid/1` predicate to prevent unacceptable rules being included into the candidates, as shown above. Because setting unreasonable conditions leads to zero rule sets generated, care should be taken when using the selection criteria and constraints for this purpose. In particular, if any of the metric predicates listed in Table 3.1 are used in defining `invalid/1`, e.g., `invalid(X) :- rule(X), metric(X, N), N < B.`, to avoid all `rule(X)` being `invalid(X)`, one should respect the conditions listed in Table 3.2.

Proposition 4. *Let the logic program*[5] *be:*

```
1 { selected(X) : valid(X) } 1.
valid(X) :- rule(X), not invalid(X).
invalid(X) :- rule(X), metric(X,N), N < B.
```

[5] `predict_class/2` and `class/1` have been omitted, and the upper bound has been changed to 1 for clarity.

Table 3.2: List of minimum and maximum values for the bounds used in defining `invalid/1`.

Metric Predicate	Relation	N	Intention	Condition
`size(X,L)`	`N > B`	`L`	Invalid if the rule is too long	$\mathtt{B} \geq min\{\mathtt{L}_1, ..., \mathtt{L}_{\lvert R\rvert}\}$
`support(X,S)`	`N < B`	`S`	Invalid if the rule has low support	$\mathtt{B} \leq max\{\mathtt{S}_1, ..., \mathtt{S}_{\lvert R\rvert}\}$
`error_rate(X,E)`	`N > B`	`E`	Invalid if the rule has high error rate	$\mathtt{B} \geq min\{\mathtt{E}_1, ..., \mathtt{E}_{\lvert R\rvert}\}$
`accuracy(X,A)`	`N < B`	`A`	Invalid if the rule has low accuracy	$\mathtt{B} \leq max\{\mathtt{A}_1, ..., \mathtt{A}_{\lvert R\rvert}\}$
`precision(X,P)`	`N < B`	`P`	Invalid if the rule has low precision	$\mathtt{B} \leq max\{\mathtt{P}_1, ..., \mathtt{P}_{\lvert R\rvert}\}$
`recall(X,R)`	`N < B`	`R`	Invalid if the rule has low recall	$\mathtt{B} \leq max\{\mathtt{R}_1, ..., \mathtt{R}_{\lvert R\rvert}\}$
`f1_score(X,F)`	`N < B`	`F`	Invalid if the rule has low F1-score	$\mathtt{B} \leq max\{\mathtt{F}_1, ..., \mathtt{F}_{\lvert R\rvert}\}$

Then, there is at least one valid rule if $\mathtt{B} \leq max(N_1, ..., N_{|R|})$.

Proof. Let $\mathtt{B} = 1 + max(\mathtt{N}_1, ..., \mathtt{N}_{|R|})$, then by line 3 (`N < B`), all rules will be `invalid`, and `valid(X)` cannot be inferred. Then, the choice rule (line 1) is not satisfied. Alternatively, let $\mathtt{B} = max(\mathtt{N}_1, ..., \mathtt{N}_{|R|})$, then there is at least one rule such that `N = B`. Since `invalid(X)` cannot be inferred for such a rule, it is `valid` and the choice rule is satisfied. □

3.4.5 Optimizing Rule Sets

Finally, we pose the rule set generation problem as a multi-objective optimization problem, given the aforementioned facts and constraints encoded in ASP. The desiderata for generated rule sets may contain multiple competing objectives. For instance, we consider a case where the user wishes to collect accurate rules that cover many instances, while minimizing the number of conditions in the set. This is encoded as a group of optimization statements:

```
% maximize accuracy and support,
% minimize the number of conditions
#maximize { A,X : selected(X), accuracy(X,A)}.
#maximize { S,X : selected(X), support(X,S)}.
#minimize { L,X : selected(X), size(X,L)}.
```

Instead of maximizing/minimizing the sums of metrics, we may wish to optimize more nuanced metrics, such as average accuracy and coverage of selected rules:

```
% maximize average accuracy and coverage
selected_rules(SR) :- SR = #count { I : selected(I) }, SR != 0.
#maximize { Ai/(S*SR)@3,I : selected(I), size(I,S),
            accuracy(I,Ai), selected_rules(SR) }.
#maximize { Sp/S@2,I : selected(I), size(I,S), support(I,Sp) }.
```

This metric can be maximized by selecting the smallest number of short and accurate rules. Similar metrics can be defined for precision-coverage,

```
% maximize average precision and coverage
#maximize { Pi/(S*SR)@3,I : selected(I), size(I,S),
            precision(I,Pi), selected_rules(SR) }.
#maximize { Sp/S@2,I : selected(I), size(I,S), support(I,Sp) }.
```

and for precision-recall.

```
% maximize average precision and recall
#maximize { Pi/(S*SR)@3,I : selected(I), size(I,S),
            precision(I,Pi), selected_rules(SR) }.
#maximize { R/S@2,I : selected(I), size(I,S), recall(I,R) }.
```

For optimization, we introduce a measure of overlap between the rules to be minimized. Intuitively, minimizing this objective should result in rule sets where rules share only a few conditions, which should further improve the explainability of the resulting rule sets. Specifically, we introduce a predicate `rule_overlap(X,Y,Cn)` to measure the degree of overlap between rules `X` and `Y`.

```
% minimize number of shared conditions between rules
rule_overlap(X,Y,Cn) :- selected(X), selected(Y), X!=Y,
    Cn = #count { Cx : Cx=Cy, condition(X,Cx), condition(Y,Cy) }.
#minimize { Cn,X : selected(X), selected(Y), rule_overlap(X,Y,Cn) }.
```

3.5 Rule Set Generation for Global and Local Explanations

In this section, we will describe how to generate global and local explanations with the rule set generation method. Guidotti et al. defined global explanation as descriptions of how the overall system works, and local explanation as specific descriptions of why a certain decision was made [20]. We shall now adopt these definitions to our rule set generation task from tree-ensemble models.

Definition 2. *A global explanation is a set of rules that approximates the overall predictive behavior of the base tree-ensemble model.*

Definition 3. *A local explanation is a set of rules that approximates the predictive behavior of the base tree-ensemble model when applied to a specific prediction instance.*

The predictive behavior in this context refers to the method by which the model makes the prediction (aggregating decision tree outputs) and the outcomes of the prediction. The differences between the global and local explanations have implications on the encoding we use for rule set generation.

Recall that we start with the candidate rule set, R, which is created by processing the tree-ensemble model. The rules in R are different between global and local explanations, even when the underlying tree-ensemble model is the same. For global explanations, we can enumerate all rules including internal nodes (Section 3.4.2) regardless of the outcomes of the rules because we are more interested in obtaining a simpler classifier with the help of constraints (Section 3.4.4) and optimization criteria (Section 3.4.5). On the other hand, for local explanations, it is necessary to consider the match between the rules' prediction and actual outcome of the tree-ensemble model as to keep the precision of explanations high.

By definition, a local explanation should describe the behavior of the model *on a single prediction instance*. Thus, we shall make the following modifications to R when generating rule sets for local explanations. We start from the candidate rule set R as in Section 3.4.2, then, for each predicted instance:

1. Identify the leaf nodes that were active during the prediction.
2. Exclude rules that did not participate in the prediction.
3. Replace the outcome of the rule with the predicted label.

After these steps, for each decision tree in a tree-ensemble model, there will be at least one rule with the outcome that is identical to the predicted label.

Proposition 5. *Let K be the number of decision trees in a tree-ensemble model, and let each of the trees have the height of h. Then, the size of the candidate rule set R, after the modifications for local explanations, is $K \times h$ for the all rules case and K for the reduced size case (only rules containing leaf nodes are considered).*

Proof. For the all rules case, the longest rule has h conditions, and additionally there are shorter rules with $1, 2, ..., h-1$ conditions. Thus, if there are K decision trees and all trees have the same height h, there will be $K \times h$ rules. For the reduced size case, since there is exactly one leaf node per tree responsible for the prediction, there will be K rules. □

3.6 Experiments

We evaluate our rule set generation framework on several public datasets and compare the performance to existing methods, including rule-based classifiers.

3.6.1 Experimental Setup

We used in total 14 publicly available datasets, where except for the *adult*[6] dataset, all datasets were taken from the UCI Machine Learning Repository[7] [46]. We included 3

[6] https://github.com/propublica/compas-analysis
[7] https://archive.ics.uci.edu/ml/index.php

datasets (*adult*, *credit german*, *compas*) for comparison because they were widely used in local explainability literature. The *adult* dataset is actually a subset of the *census* dataset, but we included the former for consistency with existing literature and the latter for demonstrating applicability of our approach to larger datasets. Other datasets were chosen since they were used in previous works in inductive logic programming. The summary of these datasets is shown in Table 3.3.

We used *clingo* 5.4.0[8] [14] for answer set programming, and set the time-out to 600 seconds. We used RIPPER implemented in Weka [47] and an open-source implementation of RuleFit[9] where Random Forest was selected as the rule generator, and scikit-learn[10] [48] for general machine learning functionalities. Our experimental environment is a desktop machine with Ubuntu 18.04, Intel Core i9-9900K 3.6GHz (8 cores/16 threads) and 64 GB RAM. For reproducibility, all source codes for the implementation, experiments and preprocessed datasets are available from our GitHub repository[11].

Unless noted otherwise, all experimental results reported here were obtained with 5-fold cross validation, with hyperparameter optimization in each fold. We used *optuna* ([49]) for hyperparameter optimization (parameter settings shown in Table 3.4). To obtain results in a reasonable amount of time, we used the reduced size rule extraction method, where only complete rules leading to the leaf nodes were considered, and the internal nodes were ignored.

To evaluate the performance of the extracted rule sets, we implemented a naive rule-based classifier, which is constructed from the rule sets extracted with our method. In this classifier, we apply the rules sequentially to the validation dataset and if all conditions within a rule are true for an instance in the dataset, the consequent of the rule is returned as the predicted class. More formally, given a set of rules $R_s \subset R$ with cardinality $|R_s|$ that shares the same consequent $class(Q)$, we represent this rule-based classifier as the disjunction of antecedents of the rules:

$$class(Q) \Leftarrow body(R_1) \vee body(R_2) \vee ... \vee body(R_r) \text{ where } 1 \leq r \leq |R_s|$$

[8] https://potassco.org/clingo/
[9] https://github.com/christophM/rulefit
[10] https://scikit-learn.org/
[11] https://github.com/atakemura/treetap

Table 3.3: Datasets used in the experiments.

Dataset	# *data*[a]	# *feature*[b]	Ratio of label = 1	Meaning of $y = 1$
adult	48,842	12 (8)	0.24	income > 50k
autism	704	20 (18)	0.27	screening result
breast	699	9 (9)	0.34	malignant
cars	1,728	6 (6)	0.30	acceptable condition
census	299,285	40 (33)	0.06	income > 50k
compas	7,214	11 (7)	0.28	2 year recidivism
credit australia	690	14 (8)	0.44	application accepted
credit german	1,000	20 (13)	0.30	good creditor
credit taiwan	30,000	23 (10)	0.22	payment next month
heart	270	13 (8)	0.44	disease present
ionosphere	351	34 (0)	0.64	good radar return
kidney	400	24 (13)	0.62	chronic disease
krvskp	3,196	36 (36)	0.52	white can win
voting	435	16 (16)	0.61	democrat

[a]Number of data points (rows).
[b]Number of features (columns). Number of categorical features is shown in parentheses.

For a given data point, it is possible that there are no rules applicable, and in such cases the most common class label in the training dataset is returned.

3.6.2 Evaluating Global Explanations

Let us recall that the purpose of generating global explanations is to provide the user with a simpler model of the original complex model. Thus, we introduce proxy measures to evaluate (1) the degree to which the model is simplified, by the number of extracted rules and (2) the number of conditions (literals) in rules, and (3) the relevance of the extracted rules, by comparing classification performance metrics against the original model.

We conducted the experiment in the following order. First, we trained Decision Tree, Random Forest and LightGBM on the datasets in Table 3.3.[12] We then applied our rule set generation method to the trained tree-ensemble models. Finally, we constructed a naive rule-based classifier using the set of rules extracted in the previous

[12]Details on hyperparameter optimization are available online on our GitHub repository (https://github.com/atakemura/treetap).

Table 3.4: Search space definition for hyperparameter optimization

Parameter	Type	Value Range	Step
Decision Tree			
max_depth	integer	[2, 9]	
min_samples_leaf	float	[1e-3, 0.2]	
min_weight_fraction_leaf	float	[0, 0.5]	0.01
criterion	categorical	[gini, entropy]	
Random Forest			
n_estimators	integer	[50, 500]	10
max_depth	integer	[2, 9]	
min_samples_leaf	float	[1e-3, 0.2]	
min_weight_fraction_leaf	float	[0.0, 0.5]	0.01
criterion	categorical	[gini, entropy]	
LightGBM			
objective	categorical	binary	
metric	categorical	binary logloss	
num_boost_round	integer	500	
early_stopping	integer	30	
learning_rate	float	[0.01, 0.2]	loguniform
max_depth	integer	[2, 9]	
num_leaves	integer	[2, 100]	
min_data_in_leaf	integer	[1, 500]	10
min_child_weight	0.001, 10	loguniform	
feature_fraction	float	[0.05, 1.0]	uniform
subsample	float	[0.2, 1.0]	uniform
subsample_freq	int	[1, 20]	
lambda_l1	[1e-5, 10]	loguniform	
lambda_l2	[1e-5, 10]	loguniform	
RuleFit			
rule_generator	categorical	random forest	
memory_parameter	float	[0.0, 1.0]	0.1
lin_standardise	boolean		
lin_trim_quantile	boolean		
RIPPER			
num_folds	integer	[2, 5]	1
prune	boolean		
no_error_check	boolean		

step, and calculated performance metrics on the validation set. This process was repeated in a 5-fold stratified cross validation setting to estimate the performance. We compare the characteristics of our approach against the known methods RIPPER and RuleFit.

We used the following selection criteria to filter out rules that were considered to be undesirable; for example, those rules with low accuracy or low coverage. We used the same set of selection criteria for all datasets, irrespective of underlying label distribution or learning algorithms. When the candidate rules violate any one of those criteria, they are excluded from the candidate rule set, which means that in the worst case where all the candidate rules violate at least one criterion, this encoding will result in an empty rule set (see Section 3.4.4).

```
% exclude long rules
invalid(I) :- size(I,S), S > 10, rule(I).
% exclude inaccurate rules
invalid(I) :- error_rate(I,E), E > 70, rule(I).
% exclude low precision rules
invalid(I) :- precision(I,P), P < 2, rule(I).
% exclude low recall rules
invalid(I) :- recall(I,R), R < 2, rule(I).
% exclude low coverage rules
invalid(I) :- support(I,Sp), Sp < 2, rule(I).
```

Another scenario in which our method will produce an empty rule set is when the tree-ensemble contains only "leaf-only" or "stump" trees, that have one leaf node and no splits. In this case, we have no split information to create candidate rules; thus, an empty rule set is returned to the user. This is often caused by inadequate setting of hyperparameters that control the growth of the trees, especially when using imbalanced datasets. It is however outside the scope of this paper, and we will simply note such cases (empty rule set returned) in our results without further consideration.

Number of Rules

The average size of candidate rule sets and average size of generated rule set size are shown in Table 3.5. Rule set size of 1 means that the rule set contains a single rule only.

As one might expect, the Decision Tree consistently has the smallest candidate rule set, but in some cases the Random Forest produced considerably more candidate rules than the LightGBM, e.g., *cars, compas*. Our method can produce rule sets which are significantly smaller than the original model, based on the comparison between the sizes of the candidate rule set $|R|$ and resulting rule sets.

We will now compare our method to the two benchmark methods, RuleFit and RIPPER. The average size of generated rule sets is shown in Table 3.6. RuleFit includes original features (called linear terms) as well as conditions extracted from the tree-ensembles in the construction of a sparse linear model, that is to say, the counts in Table 3.6 may be inflated by the linear terms. On the other hand, the output from RIPPER only contains rules, and RIPPER has rule pruning and rule set optimization to further reduce the rule set size. Moreover, RIPPER has direct control over which conditions to include into rules, whereas our method and RuleFit rely on the structure of the underlying decision trees to construct candidate rules.

Our method consistently produced smaller rule sets compared to RuleFit and RIPPER, although the difference between our method and RIPPER was not as pronounced when compared to the difference between our method and RuleFit. RuleFit produced the largest number of rules compared with other methods, although they were much smaller than the original Random Forest models (Table 3.5 and 3.5).

Number of Conditions

Another proxy measure for studying the complexity of the extracted rule sets is the number of conditions (literals) in the individual rules. Depending on the rule induction algorithm, it is possible to create rule sets with as few rules as possible, while including many conditions into rules. On the other hand, it is also possible to create many rules with few conditions. We evaluate this proxy measure from two perspectives: (1) total number of conditions in rule sets, which can be seen as another proxy for the overall complexity of the rule sets, and (2) average number of conditions in rules, which can be seen as a proxy for the readability of individual rules in rule sets.

Table 3.7 shows the total number of conditions in rule sets, averaged over 5 folds. Overall, the decision tree-based algorithms (decision tree, random forest and LightGBM) combined with our ASP encoding produces rule sets with small number of conditions

Table 3.5: Average number of candidate rules ($|R|$), size of the generated rule sets (# rule), averaged over 5 folds. (Global Explanations)

	Decision Tree+ASP		Random Forest+ASP		LightGBM+ASP	
Dataset	$\|R\|$	# rule	$\|R\|$	# rule	$\|R\|$	# rule
adult	104.6	1.0	1,774.4	1.4	4,227.6	1.4
autism	2.0	1.0	833.6	1.0	2.0	1.0
breast	17.2	1.0	749.6	1.2	409.4	1.0
cars	41.4	1.0	7,502.0	1.0	1,308.0	1.0
census	81.8	1.0	585.2	1.0	9,533.0	1.0
compas	57.4	1.0	6,702.2	1.0	1,047.6	1.0
credit australia	4.2	1.0	1,832.4	1.2	350.4	1.0
credit german	44.0	1.0	5,638.8	2.0	628.2	1.2
credit taiwan	42.2	1.0	3,975.0	1.0	2,854.2	1.4
heart	6.2	1.2	1,553.6	1.0	213.0	1.4
ionosphere	8.4	1.0	682.4	1.0	315.8	1.0
kidney	7.4	1.0	639.4	1.0	356.6	1.2
krvskp	33.6	1.0	4,122.4	1.0	2,354.6	1.0
voting	10.2	1.0	1,306.2	1.0	153.6	1.0

Table 3.6: Average size of the generated rule sets (# rule), averaged over 5 folds. (Global Explanations)

Dateset	DT[a]+ASP	RF[b]+ASP	LGBM[c]+ASP	RuleFit	RIPPER
adult	**1.0**	1.4	1.4	356.8	24.4
autism	**1.0**	**1.0**	**1.0**	3.2	2.0
breast	**1.0**	1.2	**1.0**	148.6	14.8
cars	**1.0**	**1.0**	**1.0**	481.2	28.0
census	**1.0**	**1.0**	**1.0**	895.6	60.6
compas	**1.0**	**1.0**	**1.0**	176.4	10.4
credit australia	**1.0**	1.2	**1.0**	58.2	5.4
credit german	**1.0**	2.0	1.2	438.4	4.2
credit taiwan	**1.0**	**1.0**	1.4	113.2	7.2
heart	1.2	**1.0**	1.4	400.4	5.6
ionosphere	**1.0**	**1.0**	**1.0**	612.2	5.0
kidney	**1.0**	**1.0**	1.2	79.2	4.6
krvskp	**1.0**	**1.0**	**1.0**	317.0	17.6
voting	**1.0**	**1.0**	**1.0**	79.8	3.4

[a]DT=Decision Tree
[b]RF=Random Forest.
[c]LGBM=LightGBM.

Table 3.7: Total number of conditions in rule sets, averaged over 5 folds. (Global Explanations)

Dateset	DT[a]+ASP	RF[b]+ASP	LGBM[c]+ASP	RuleFit	RIPPER
adult	**4.0**	7.0	7.0	1143.4	108.4
autism	**1.0**	**1.0**	**1.0**	4.4	**1.0**
breast	5.8	6.0	**1.0**	427.8	19.8
cars	6.4	5.2	**1.0**	1541.4	101.8
census	8.2	5.4	**1.8**	2307.4	381.2
compas	3.6	5.8	**2.4**	559.6	27.6
credit australia	**2.2**	3.2	2.4	156.4	9.8
credit german	4.0	18.0	**3.4**	1262.6	8.4
credit taiwan	**4.6**	6.2	5.6	317.6	18.0
heart	2.2	**2.0**	3.2	1057.4	13.4
ionosphere	**4.2**	4.8	4.6	1408.8	6.2
kidney	**2.4**	**2.4**	4.0	221.6	7.2
krvskp	6.0	7.8	**3.0**	998.6	59.2
voting	2.2	**1.0**	1.8	205.4	5.8

[a]DT=Decision Tree
[b]RF=Random Forest.
[c]LGBM=LightGBM.

Table 3.8: Average number of conditions for each rule in rule sets, averaged over 5 folds. (Global Explanations)

Dateset	DT[a]+ASP	RF[b]+ASP	LGBM[c]+ASP	RuleFit	RIPPER
adult	4.0	4.8	5.0	**3.2**	4.4
autism	1.0	1.0	1.0	1.3	**0.5**
breast	5.8	5.0	**1.0**	3.5	1.3
cars	6.4	5.2	**1.0**	3.3	3.4
census	8.2	5.4	**1.8**	2.6	6.3
compas	3.6	5.8	**2.4**	3.2	2.6
credit australia	2.2	3.0	2.4	2.7	**1.6**
credit german	4.0	9.0	2.4	3.0	**2.0**
credit taiwan	4.6	6.2	4.0	2.8	**2.5**
heart	**1.8**	2.0	2.0	2.0	2.1
ionosphere	4.2	4.8	4.6	2.5	**1.1**
kidney	2.4	2.4	3.4	2.9	**1.5**
krvskp	6.0	7.8	**3.0**	3.2	3.3
voting	2.2	**1.0**	1.8	2.4	1.3

[a]DT=Decision Tree
[b]RF=Random Forest.
[c]LGBM=LightGBM.

than the benchmark methods. In particular, RuleFit produced rule sets with much larger number of conditions in all datasets. The results for RIPPER were mixed in this case; while the number of conditions in rule sets were often similar to our methods, in other cases it produced rule sets with large number of conditions (e.g., *census* and *cars* datasets).

Table 3.8 shows the average number of conditions per rule in rule sets, averaged over 5 folds. Contrary to the previous results (Tables 3.6, 3.7), the results are similar among different methods. When compared to the results for the total number of rules and conditions in rule sets, we see that RuleFit produces a large number of rules, but individual rules are short (3 conditions or fewer, on average). RIPPER produced, in some cases, the smallest rules in terms of average number of conditions per rule. However, for RIPPER, it should be noted that the figures quoted in Table 3.8 includes rules with 0 conditions (e.g., a 'default' rule with empty body, '=> class 0'), so the average number of conditions in bodies is likely to be biased towards lower values. The figures for the ASP-based methods and RuleFit do not include the 'default' rule, as in

RIPPER. As for our method, the maximum number of conditions included in each of the rules is actually capped by the maximum depth of the decision trees in the ensemble. It can additionally be affected by the ASP encoding, if the user decides to include it in the selection criteria, however, it was not the case in this experiment. Note also that the maximum depth in the decision tree training parameter means the maximum possible depth to which the tree may be grown, and it is not a requirement for the tree induction algorithm to actually grow to that depth. In our experiments, the maximum depth parameter in the hyperparameter selection was set to much higher values than the values in Table 3.8, thus either the hyperparameter tuning algorithm or the tree-ensemble learning algorithm could have decided that deeper trees were not necessary.

Relevance of Rules

To quantify the relevance of the extracted rules, we measured the ratio of performance metrics using the naive rule-based classifier by 5-fold cross validation (Table 3.9 and 3.10). Performance ratio of less than 1.0 means that the rule-based classifier performed worse than the original classifier (LightGBM and Random Forest), whereas performance ratio greater than 1.0 means the rule set's performance is better than the original classifier. We used a version of the ASP encoding shown in Section 3.4.5 where the accuracy and coverage are maximized. RIPPER was excluded from this comparison because it has a built-in rule generation and refinement process, and it does not have a base model, whereas our method and RuleFit use variants of tree-ensemble models as base models.

From Table 3.9 we observe that in terms of accuracy, RuleFit generally performs as well as, or marginally better than, the original Random Forest. On the other hand, although our method can produce rule sets that are comparable in performance against the original model, they do not produce rules that perform significantly better. With Decision Tree and Random Forest, the generated rule sets perform much worse than the original model, e.g., in *kidney*, *voting*. The LightGBM+ASP combination resulted in the second-best performance overall, where the resulting rules' performances were arguably comparable (0.8-0.9 range) to the original model with a few exceptions (e.g., *census* F1-score) where the performance ratio was about half of the original. While

Table 3.9: Average ratio of rule-based classifier's performance vs. original tree-ensembles, averaged over 5 folds. Accuracy and F1-score ratio. (Global Explanations)

	Accuracy ratio[a]				F1-score ratio			
Dataset	DT[b]	RF[c]	LGBM[d]	RuleFit	DT	RF	LGBM	RuleFit
adult	0.92	0.93	0.94	**1.01**	0.34	0.63	0.78	**1.11**
autism	**1.00**	**1.00**	**1.00**	**1.00**	1.00	**1.01**	1.00	**1.01**
breast	0.91	0.96	0.95	**0.99**	0.80	0.93	0.92	**0.98**
cars	0.80	0.81	0.52	**1.01**	0.41	0.49	0.44	**1.02**
census	0.96	1.00	0.80	**1.02**	0.21	5.17	0.40	**10.86**
compas	0.96	0.93	0.91	**1.01**	0.47	0.58	0.78	**1.08**
credit australia	0.89	0.92	0.88	**1.00**	0.76	0.86	0.79	**1.00**
credit german	0.90	0.92	0.79	**0.96**	0.44	0.86	0.85	**1.19**
credit taiwan	0.97	0.97	0.99	**1.01**	0.39	0.62	0.87	**1.11**
heart	0.88	0.90	0.99	**1.01**	0.76	0.83	0.95	**1.02**
ionosphere	0.72	0.70	0.96	**0.99**	0.85	0.83	0.97	**0.99**
kidney	0.67	0.62	0.90	**0.99**	0.81	0.77	0.92	**1.00**
krvskp	0.53	0.54	0.59	**1.02**	0.69	0.71	0.67	**1.02**
voting	0.64	0.64	0.97	**1.00**	0.78	0.79	0.97	**1.00**

[a]Performance ratio of 1 means the rule set's performance is identical to the original classifier.
[b]DT=Decision Tree+ASP.
[c]RF=Random Forest+ASP.
[d]LGBM=LightGBM+ASP.

RuleFit's performance was superior, our method could still produce rule sets with reasonable performance with much smaller rule sets that are an order of magnitude smaller than RuleFit. A rather unexpected result was that using our method (Random Forest) or RuleFit significantly improved the F1-score in the *census* dataset. In Table 3.10 we can see that recall was the major contributor to this improvement.

Changing Optimization Criteria

The definition of optimization objectives has a direct influence over the performance of the resulting rule sets, and the objectives need to be set in accordance with user requirements. The answer sets found by *clingo* with multiple optimization statements are optimal regarding the set of goals defined by the user. Instead of using accuracy, one may use other rule metrics as defined in Table 3.1 such as precision and/or recall. If there are priorities between optimization criteria, then one could use the priority

Table 3.10: Average ratio of rule-based classifier's performance vs. original tree-ensembles, averaged over 5 folds. Precision and Recall ratio. (Global Explanations)

	Precision ratio[a]				Recall ratio			
Dataset	DT[b]	RF[c]	LGBM[d]	RuleFit	DT	RF	LGBM	RuleFit
adult	**1.30**	1.00	0.86	0.94	0.22	0.69	0.74	**1.25**
autism	**1.00**	**1.00**	**1.00**	**1.00**	1.00	**1.01**	1.00	**1.01**
breast	0.97	0.97	0.96	**0.99**	0.70	0.91	0.88	**0.97**
cars	1.04	**1.06**	0.36	1.04	0.26	0.32	0.60	**0.99**
census	0.07	0.54	0.30	**0.75**	0.06	6.81	1.26	**17.14**
compas	**1.13**	0.85	0.84	0.96	0.34	0.50	0.85	**1.18**
credit australia	**1.16**	1.00	1.01	1.01	0.55	0.81	0.70	**0.99**
credit german	**0.81**	0.69	0.61	0.77	0.41	0.99	1.16	**1.54**
credit taiwan	**1.01**	0.90	0.98	0.98	0.29	0.55	0.84	**1.18**
heart	0.96	0.94	**1.08**	1.01	0.68	0.78	0.86	**1.07**
ionosphere	0.70	0.71	0.98	**1.01**	**1.09**	1.03	0.97	0.99
kidney	0.67	0.62	0.91	**1.00**	**1.04**	1.00	0.94	1.00
krvskp	0.53	0.54	0.65	**1.03**	1.00	**1.02**	0.76	1.01
voting	0.62	0.63	**1.01**	0.98	1.04	**1.05**	0.94	1.00

[a]Performance ratio of 1 means the rule set's performance is identical to the original classifier.
[b]DT=Decision Tree+ASP.
[c]RF=Random Forest+ASP.
[d]LGBM=LightGBM+ASP.

notation (`weight@priority`) in *clingo* to define them. Optimal answer sets can be computed in this way, however, if enumeration of such optimal sets is important, then one could use the `pareto` or `lexico` preference definitions provided by *asprin* [50] to enumerate Pareto optimal answer sets. Instead of presenting a single optimal rule set to the user, this will allow the user to explore other optimal rule sets.

To investigate the effect of changing optimization objectives, we changed the ASP encoding from max. accuracy-coverage to max. precision-coverage (shown in Section 3.4.4) while keeping other parameters constant. The results are shown in Table 3.11. Note that it is the ratio of precision score shown in the table, as opposed to accuracy or F1-score in the earlier tables. Here, since we are optimizing for better precision, we expect the precision-coverage encoding to produce rule sets with better precision scores than the accuracy-coverage encoding. For the Decision Tree and Random Forest + ASP, the effect was not as pronounced as we expected, but we observed noticeable differences in datasets *compas* and *credit german*. For the LightGBM+ASP combination, we observed more consistent difference, except for the *ionosphere* dataset, the encoding produced intended results in most of the datasets in this experiment.

3.6.3 Evaluating Local Explanations

The purpose of generating local explanations is to provide the user with an explanation for the model's prediction for each predicted instance. Here, we use commonly used metrics *local-precision* and *coverage* as proxy measures for the quality of the explanation.[13] The local-precision compares the (black-box) model predictions of instances covered by the local explanation and the model prediction of the original instance used to induce the local explanation. The coverage is the ratio of instances in the validation set that are covered by the local explanation. These two metrics are in a trade-off relationship, where pursuing high coverage is likely to result in low precision explanation and vice versa. Additionally, we will also compare the running time to generate the local explanation.

The experiments were carried out similarly to the global explanation evaluation, except that: (1) we replaced RIPPER and RuleFit with Anchors, (2) instead of using the

[13] In the original Anchors paper, the authors use the term *precision*, but here we add *local-* to distinguish from the more commonly used definition of precision.

Table 3.11: Average ratio of rule-based classifier's precision vs. original tree-ensembles, averaged over 5 folds. (Global Explanations)

	Decision Tree+ASP[a]		Random Forest+ASP		LightGBM+ASP	
Dataset	acc.cov[b]	prec.cov[c]	acc.cov	prec.cov	acc.cov	prec.cov
adult	**1.30**	**1.30**	1.00	**1.13**	0.86	**1.27**
autism	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
breast	**0.97**	**0.97**	0.97	**1.04**	0.96	**1.02**
cars	**1.04**	**1.04**	**1.06**	**1.06**	**0.36**	**0.36**
census	0.07	**0.24**	**0.54**	**0.54**	0.30	**0.90**
compas	1.13	**1.27**	0.85	**1.05**	0.84	**1.11**
credit australia	**1.16**	**1.16**	1.00	**1.05**	1.01	**1.04**
credit german	0.81	**1.28**	0.69	**0.92**	0.61	**0.75**
credit taiwan	1.01	**1.04**	0.90	**1.08**	0.98	**1.12**
heart	0.96	**1.13**	0.94	**1.02**	1.08	**1.12**
ionosphere	**0.70**	**0.70**	**0.71**	**0.71**	**0.98**	0.95
kidney	**0.67**	**0.67**	**0.62**	**0.62**	0.91	**0.97**
krvskp	**0.53**	**0.53**	**0.54**	**0.54**	0.65	**0.70**
voting	**0.62**	**0.62**	**0.63**	**0.63**	1.01	**1.02**

[a]Performance ratio of 1 means the rule set's precision is identical to the original classifier. Numbers are shown in bold where the performance ratio was better than more than 0.01 compared to the other encoding.

[b]acc.cov=accuracy and coverage encoding, see Section 4.

[c]prec.cov=precision and coverage encoding, see Section 4.

full validation set, we resampled the validation dataset to generate 100 instances in each cross-validation fold for each dataset to estimate the metrics, to complete the experiments in a reasonable amount of time, and (3) in the ASP encoding, we removed the rule selection criteria to avoid excluding rules that are relevant to the predicted instance. We were unable to complete Anchors experiment with the *census* dataset due to limited memory (64 GB) on our machine. For the running time comparison, we exclude all data preprocessing, training and tree processing, and focus solely on the time taken to generate local explanations.

Local-Precision, Coverage and Running Time

The average local-precision, averaged over 5 cross-validation folds, is reported in Table 3.12. Note that while Anchors has a minimum precision threshold (we used the default 0.95 setting), ours do not, and indeed we see that all Anchors explanations have higher local-precision than the threshold. The Decision Tree will always have exactly one rule that is relevant to the prediction; therefore, we expect to see exactly 1 local-precision using our method. For the Random Forest and LightGBM, our method produced local explanations with local-precision in 0.8-0.9 range for most of the datasets, but Anchors' explanations had higher local-precision in most cases.

The average coverage, averaged over 5 cross-validation folds, is reported in Table 3.13. Interestingly, when using simpler models such as the Decision Tree and Random Forest, Anchors can produce rules that have relatively high coverage, but the pattern does not hold when using a more complex model, which in our case is LightGBM. With LightGBM, our method consistently outperformed Anchors in terms of coverage in all datasets, except for the *census* dataset, which we could not run.

Finally, the average running time per instance is reported in Table 3.14. For the Decision Tree, Anchors was faster than our method, whereas for the Random Forest and LightGBM, our method was faster than Anchors in most datasets. We also note that our method has a more consistent running time of around 2 seconds across all datasets, regardless of the complexity of the underlying models, whereas Anchors' running time varies from sub-1 second to tens of seconds, depending on the dataset and model. This is likely to be caused by the differences in which these methods query or use information from the original model and generate explanations. In fact,

Table 3.12: Average local-precision of local explanations, averaged over 5 folds. The ASP encoding used was precision-coverage. (Local Explanations)

	Decision Tree		Random Forest		LightGBM	
Dataset	Ours	Anchors[a]	Ours	Anchors	Ours	Anchors
adult	**1.00**	0.98	0.77	**0.99**	0.87	**1.00**
autism	**1.00**	**1.00**	0.78	**1.00**	**1.00**	**1.00**
breast	**1.00**	**1.00**	0.71	**1.00**	0.91	**0.99**
cars	**1.00**	0.99	0.90	**1.00**	0.72	**1.00**
census	1.00	n/a	0.99	n/a	0.94	n/a
compas	**1.00**	0.99	0.78	**0.98**	0.89	**1.00**
credit australia	**1.00**	**1.00**	0.74	**0.99**	0.90	**1.00**
credit german	**1.00**	0.98	0.92	**1.00**	0.86	**1.00**
credit taiwan	**1.00**	0.98	0.92	**0.99**	0.98	**1.00**
heart	**1.00**	**1.00**	0.64	**0.99**	0.90	**1.00**
ionosphere	**1.00**	**1.00**	0.81	**1.00**	0.93	**1.00**
kidney	**1.00**	0.99	0.80	**1.00**	0.89	**1.00**
krvskp	**1.00**	0.99	0.80	**1.00**	0.75	**1.00**
voting	**1.00**	0.99	0.91	**1.00**	**0.99**	0.98

[a] n/a indicates where Anchors ran out of memory on our system.

a significant amount of time is spent in tree processing in our method, whereas in Anchors the search process is often the most time-consuming step. Nonetheless, this comparative experiment demonstrated that our method can produce local explanations in a matter of seconds even when the underlying tree-ensemble is large.

Number of Conditions in Local Explanations

For completeness, we report the number of conditions in local explanations in this section, calculated in a similar manner to the corresponding metric in global explanations. For local explanations, the number of rules provided as an explanation is usually 1, i.e., there is usually only a single rule given as an explanation. Thus, we focus on the average length (number of conditions in the body) of the rules, instead of the total number of conditions or the number of rules, as in global explanations. The results are shown in Table 3.15.

For decision tree and random forest, Anchors produced smaller rules overall compared to our methods. It is interesting that this trend is completely reversed for

Table 3.13: Average coverage of local explanations, averaged over 5 folds. The ASP encoding used was precision-coverage. (Local Explanations)

	Decision Tree		Random Forest		LightGBM	
Dataset	Ours	Anchors[a]	Ours	Anchors	Ours	Anchors
adult	0.10	**0.26**	0.14	**0.36**	**0.79**	0.01
autism	**0.62**	0.11	**0.17**	0.08	**0.62**	0.11
breast	0.35	**0.38**	**0.32**	0.28	**0.60**	0.04
cars	0.17	**0.21**	0.06	**0.21**	**0.54**	0.01
census	0.20	n/a	0.27	n/a	**0.94**	n/a
compas	**0.10**	0.09	**0.20**	0.12	**0.55**	0.01
credit australia	0.32	**0.48**	**0.25**	0.21	**0.33**	0.02
credit german	0.17	**0.24**	0.03	**0.07**	**0.49**	0.01
credit taiwan	0.18	**0.61**	0.07	**0.61**	**0.63**	0.01
heart	0.24	**0.37**	**0.29**	0.17	**0.29**	0.03
ionosphere	**0.35**	0.04	**0.39**	0.02	**0.41**	0.02
kidney	**0.32**	0.11	**0.35**	0.05	**0.51**	0.02
krvskp	0.13	**0.17**	**0.14**	0.13	**0.51**	0.01
voting	0.33	**0.50**	**0.42**	**0.42**	**0.46**	0.04

[a] n/a indicates where Anchors ran out of memory on our system.

Table 3.14: Average running time per instance, in seconds, averaged over 5 folds. The ASP encoding used was precision-coverage. (Local Explanations)

	Decision Tree		Random Forest		LightGBM	
Dataset	Ours	Anchors[a]	Ours	Anchors	Ours	Anchors
adult	**1.73**	1.86	**1.90**	4.35	**2.09**	27.98
autism	1.71	**0.84**	**1.80**	8.54	1.86	**1.00**
breast	1.70	**1.04**	**1.81**	16.33	**2.13**	8.16
cars	1.70	**0.17**	**2.00**	3.12	1.92	**1.16**
census	**1.76**	n/a	**2.04**	n/a	**2.18**	n/a
compas	1.72	**0.23**	**2.02**	2.55	1.99	**1.27**
credit australia	1.70	**0.25**	**1.89**	12.78	**1.85**	2.70
credit german	1.73	**1.02**	**1.95**	9.44	**1.94**	7.52
credit taiwan	1.72	**1.57**	**1.95**	4.13	**2.19**	45.48
heart	1.62	**0.25**	**2.17**	12.37	**1.73**	2.01
ionosphere	1.62	**1.26**	**1.90**	19.09	**1.73**	13.17
kidney	1.70	**0.85**	**1.79**	23.01	**1.89**	10.84
krvskp	1.70	**1.16**	**1.97**	10.69	**2.12**	12.03
voting	1.70	**0.33**	**1.86**	18.18	**1.88**	4.65

[a] n/a indicates where Anchors ran out of memory on our system.

Table 3.15: Average number of conditions per explanation, averaged over 5 folds. The ASP encoding used was precision-coverage. (Local Explanations)

	Decision Tree		Random Forest		LightGBM	
Dataset	Ours	Anchors[a]	Ours	Anchors	Ours	Anchors
adult	8.65	**3.31**	6.37	**3.07**	**4.94**	92.88
autism	**1.00**	1.55	3.75	**2.45**	**1.00**	1.55
breast	4.32	**1.48**	3.91	**2.03**	**1.41**	30.51
cars	4.03	**2.20**	6.28	**2.25**	**1.17**	20.46
census	**8.08**	n/a	**7.86**	n/a	**7.57**	n/a
compas	4.99	**3.80**	3.74	**2.88**	**3.34**	21.92
credit australia	1.87	**1.10**	3.17	**2.48**	**2.55**	39.13
credit german	4.32	**3.34**	6.32	**4.83**	**2.56**	54.22
credit taiwan	5.69	**1.90**	7.47	**1.57**	**3.99**	141.25
heart	2.60	**1.50**	**2.22**	2.76	**2.62**	25.93
ionosphere	3.63	**2.78**	3.95	**2.95**	**3.50**	31.48
kidney	2.84	**1.75**	2.61	**2.31**	**2.04**	41.18
krvskp	4.78	**2.62**	6.11	**3.78**	**2.52**	63.89
voting	2.45	**1.22**	2.27	**2.23**	**2.06**	21.08

[a] n/a indicates where Anchors ran out of memory on our system.

LightGBM, where our method could produce much more concise rules compared to Anchors, which sometimes produced rules with over 100 conditions. This result for LightGBM also highlights the difference between our method and Anchors, in the sense that, in our method, the search space is constrained by the structures of the decision trees, so the maximum number of conditions is capped at the maximum depth of the decision trees, whereas in Anchors, the algorithm can add as many conditions as necessary to achieve the precision guarantee set by the user.

Summary of Experiments

To conclude the experimental section, we summarize the main results obtained in this section. For global explanations, we analyzed (1) the average size of generated rule sets and compared it against known methods, as a proxy measure for the degree of simplifications, (2) the relative performance of the rule sets and compared it against known methods, as a proxy measure for the relevance of the explanations, and (3) the effect of modifying the ASP encoding on the precision metric of the explanations. Overall, our method was shown to be able to produce smaller rule sets compared to the known methods, however, in terms of the relevance of the rules, RuleFit performed better in most cases, demonstrating the trade-off relationship between the complexity of the explanations and performance.

For local explanations, we compared (1) local-precision, (2) coverage (3) running time and (4) number of conditions of our method against Anchors. In terms of local-precision, although our method could produce explanations with reasonably high precision (0.8-0.9 range), Anchors performed better overall. As for coverage, we found that explanations generated by our method can cover more examples for tree-ensemble. Regarding running time, our method had a consistent running time of around 2 seconds, whereas the running time of Anchors varied between datasets. The experiments for local explanations also highlights the differences between our method and Anchors: while Anchors can produce high-precision rules, our method has advantage in terms of memory requirement and consistent running time.

3.7 Conclusion

In this work, we presented a method for generating rule sets as global and local explanations from tree-ensemble models using pattern mining techniques encoded in ASP. Unlike other explanation methods that focus exclusively on either global or local explanations, our two-step approach allows us to handle both global and local explanation tasks. We showed that our method can be applied to two well-known tree-ensemble learning algorithms, namely Random Forest and LightGBM. Evaluation on various datasets demonstrated that our method can produce explanations with good quality in a reasonable amount of time, compared to existing methods.

Adopting the declarative programming paradigm with ASP allows the user to take advantage of the expressiveness of ASP in representing constraints and optimization criteria. This makes our approach particularly suitable for situations where fast prototyping is required, since changing the constraint and optimization settings require relatively low effort compared to specialized pattern mining algorithms. Useful explanations can be generated using our approach, and combined with the expressive ASP encoding, we hope that our method will help the users of tree-ensemble models to better understand the behavior of such models.

A limitation of our method in terms of scalability is the size of search space, which is exponential in the number of valid rules. When the number of candidate rules is large, we suggest using stricter individual rule constraints on the rules, or reducing the maximum number of rules to be included into rule sets (Section 3.4.4), to achieve reasonable solving time.

There are a number of directions for further research. First, while the current work did not modify the conditions in the rules in any way, rule simplification approaches could be incorporated to remove redundant conditions. Second, we could extend the current work to support regression problems. More generally, in the future, we plan to explore how ASP and modern statistical machine learning could be integrated effectively to produce more interpretable machine learning systems.

4

Differentiable Supported Model Computation

4.1 Introduction

With the recent interest in neuro-symbolic approaches, performing logical inference with linear algebraic methods has been studied as an attractive alternative to symbolic methods [51, 52]. Prior implementations of neuro-symbolic systems provided interfaces for the symbolic reasoning engines to handle the outputs from the neural networks as neural predicates [53, 54]. However, more direct realization of logic programming in continuous domain remains an open challenge.

A key component in a typical deep neural network is a set of parameters stored in multidimensional tensors. Thus, performing logical inference in vector spaces allows us to work with a common type of algebraic objects, which may facilitate neuro-symbolic integration. While there are many possible ways to achieve neuro-symbolic integration, in our view, there are two main requirements, one from the neuro-symbolic

perspective, and the other from the logic programming perspective. (i) The inference step should be able to handle continuous inputs and support differentiable operations in continuous domain, or an interface should be provided where the results of symbolic computation can be translated into continuous domain for the neural networks. (ii) The semantics of logic programming should be preserved even in continuous domains, and non-monotonic semantics are preferred over the classical semantics for supporting commonsense reasoning.

Matrix representations of normal logic programs and a linear algebraic method for computing the stable models were proposed by Sakama et al. [52]. Using an alternative matrix representation, Sato et al. computed supported models in vector spaces via 3-valued completion models of normal logic programs [55]. While the aforementioned methods would allow one to compute models under non-monotonic semantics in vector spaces, they use non-differentiable operations that do not use the gradient-information. More recently, gradient-based search methods have been proposed for SAT [56], supported and stable model computation [51]. Aspis et al.'s method uses a matrix representation of the program reduct, the Newton's method for root finding for finding fixed points, and a parameterized sigmoid function for thresholding. Compared to symbolic local search methods that flip one atom at a time [57], matrix- and gradient-based methods update all assignments simultaneously in continuous domain, which may reduce the number of restarts compared to discrete value search.

In the context of gradient-based search, many variations are possible for each component. In this work, we build upon previous works [56, 51] by presenting an alternative differentiable method for efficiently computing supported models of normal logic programs in continuous vector spaces. Our main contributions are:

- Presenting an alternative method for embedding logic programs into matrices, and designing an almost everywhere differentiable thresholding function.
- Introducing a loss function with regularization terms for computing supported models, and integrating various gradient update strategies.
- Demonstrating with a help of systematic performance evaluation on a range of programs, that by selecting appropriate components, it is possible to achieve much higher performance and stability than the existing method.

The structure of this section is as follows: Section 4.2 covers the necessary background and definitions. Section 4.3 presents a method for representing logic programs with matrices. Section 4.4 introduces the thresholding function, loss function, and the gradient-based search algorithm for supported models. Section 4.5 presents the results of experiments designed to test the ability of the algorithm. Finally, Section 4.6 presents the conclusion.

4.2 Background

For a general background material on logic programming, readers are referred to Section 2.1, since this section only covers materials that are relevant to this chapter.

We consider a language $\mathcal{L}$ that contains a finite set of propositional variables defined over a finite alphabet and the logical connectives $\neg$, $\wedge$, $\vee$ and $\leftarrow$. The *Herbrand base*, B_P, is the set of all propositional variables in a logic program P.

A *definite program* is a set of *rules* of the form (4.1) or (4.2), where h and b_i are propositional variables (*atoms*) in $\mathcal{L}$. We refer to (4.2) as an *OR-rule*, which is a shorthand for m rules: $h \leftarrow b_1, h \leftarrow b_2, \ldots, h \leftarrow b_m$. For each rule r we define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$. A rule r is a *fact* if $body(r) = \emptyset$.

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_m (m \geq 0) \tag{4.1}$$

$$h \leftarrow b_1 \vee b_2 \vee \cdots \vee b_m (m \geq 0) \tag{4.2}$$

A *normal program* is a set of rules of the form (4.3) where h and b_i are propositional variables in $\mathcal{L}$.

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \neg b_{l+2} \wedge \cdots \wedge \neg b_m (m \geq l \geq 0) \tag{4.3}$$

We refer to the positive and negative occurrences of atoms in the body as $body^+(r) = \{b_1, \ldots, b_l\}$ and $body^-(r) = \{b_{l+1}, \ldots, b_m\}$, respectively. A normal program is a definite program if $body^-(r) = \emptyset$ for every rule $r \in P$.

An *Herbrand interpretation* I, of a normal program P is a subset of B_P. A *model* M of P is an interpretation of P where for every rule $r \in P$ of the form (4.3), $body^+(r) \subseteq M$

and $body^-(r) \cap M = \emptyset$ imply $h \in M$. A program is called *consistent* if it has a model. A *supported model* M is a model of P where for every $p \in M$ there exists a rule $r \in P$ such that $p = h$, $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$ [8, 10].

As we shall show later, in this work we transform normal logic programs into definite programs for searching supported models. Thus, we use the following definition of the immediate consequence operator T_P. $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is a function on Herbrand interpretations. For a definite program P, we have: $T_P(I) = \{h | h \leftarrow b_1 \wedge \cdots \wedge b_m \in P \text{ and } \{b_1, \ldots, b_m\} \subseteq I\} \cup \{h | h \leftarrow b_1 \vee \cdots \vee b_m \in P \text{ and } \{b_1, \ldots, b_m\} \cap I \neq \emptyset\}$. It is known that a supported model M of a program P is a fixed point of T_P, i.e. $T_P(M) = M$[10].

Definition 4 (Singly-Defined (SD) Program)**.** *A normal program P is an SD-program if $head(r_1) \neq head(r_2)$ for any two rules r_1 and r_2 $(r_1 \neq r_2)$ in P.*

Any normal program P can be converted into an SD-program P' in the following manner. If there are more than one rule with the same head ($h \leftarrow body(r_1), \ldots, h \leftarrow body(r_k)$, where $k > 1$), then replace them with a set of new rules $\{h \leftarrow b_1 \vee ... \vee b_k, b_1 \leftarrow body(r_1), ..., b_k \leftarrow body(r_k)\}$ containing new atoms $\{b_1, ..., b_k\}$. This is a stricter condition than the *multiple definitions condition* (*MD-condition*) [52]: for any two rules r_1 and r_2 in P, (i) $head(r_1) = head(r_2)$ implies $|body^+(r_1)| \leq 1$ and $|body^+(r_2)| \leq 1$, and (ii) $body^-(r_1) \cap body^-(r_2) \neq \emptyset$ implies $|body^+(r_1)| \leq 1$ and $|body^+(r_2)| \leq 1$. All SD-programs satisfy the MD condition. We shall assume all programs in this work are SD-programs.

Given a normal program P, it is transformed into a definite program by replacing the negated literals in rules of the form (4.3) and rewriting:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \overline{b}_{l+1} \wedge \overline{b}_{l+2} \wedge \cdots \wedge \overline{b}_m (m \geq l \geq 0) \tag{4.4}$$

where $\overline{b}_i$ are new atoms associated with the negated b_i. A collection of rules of the form (4.4) is referred to as the *positive form* P^+ where $B_{P^+} = B_P \cup \{\overline{a} \mid a \in B_P\}$. For transformed rules of the form (4.4), we refer to $\{b_1, \ldots, b_l\}$ as the *positive part* and $\{\overline{b}_{l+1}, \ldots, \overline{b}_m\}$ as the *negative part*. After transformation, the program should contain rules of the forms (4.1), (4.2), or (4.4). By an interpretation I^+ of P^+, we mean any set of atoms $I^+ \subseteq B_{P^+}$ that satisfies the condition for any atom $a \in B_{P^+}$, precisely one of either a or $\overline{a}$ belongs to I^+.

4.3 Representing Logic Programs with Matrices

4.3.1 Relationship between Positive Forms and Supported Models

Consider a program $p \leftarrow \neg p$, and its positive form $p \leftarrow \overline{p}$. P^+ is a definite program, but it has no supported models in this case due to the restriction we place on the interpretation: if $p \in I^+$ then $\overline{p} \notin I^+$ and vice versa. Then in this case, the implication is that there are no fixed points of T_{P^+} for P^+ that satisfy the condition $p \in I^+$ iff $\overline{p} \notin I^+$. On the other hand, when a model M of P exists, we can show that the corresponding M^+ is a model of P^+.

Proposition 6. *Let P be a normal program, and let P^+ be its positive form. If M is a model of P, then $M' = M \cup \{\overline{a} \mid a \in B_{P^+} \setminus M\}$ is a model of P^+. Conversely, if M^+ is a model of P^+, then $M^+ \cap B_P$ is a model of P.*

Proof. Follows from the definition of M' and M^+. Consider M'. Since $a \notin M'$ if $\overline{a} \in M'$ and vice versa, for each rule $r \in P^+$, $body(r) \subseteq M'$ implies $head(r) = a \in M'$. Thus, M' is a model of P+. Now consider M^+. Let $K = M^+ \cap B_P$. Given that M^+ is a model of P^+ and $a \in K$ if $a \in M^+$, for each rule $r \in P$, $body^+(r) \subseteq K$ and $body^-(r) \cap K = \emptyset$ implies $head(r) = a \in K$. Thus, K is a model of P. □

Proposition 7. *Let M be a supported model of P, and put $M' = M \cup \{\overline{a} \mid a \in B_{P^+} \setminus M\}$. Then, $T_{P^+}(M') = M$.*

Proof. Suppose $a \in M$. Since M is a supported model, there exists a rule $r \in P$ such that $head(r) = a$, $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$. Correspondingly, there exists a rule $r' \in P^+$ such that $head(r') = a$, $body^+(r') \subseteq M'$ and $body^-(r') \subseteq M'$. That is, $a \in T_{P^+}(M')$. Hence, $M \subseteq T_{P^+}(M')$.

Conversely, suppose $a \in T_{P^+}(M')$. Then, there exists a rule $r' \in P^+$ such that $head(r') = a$ and $body(r') \subseteq M'$. Since M' is a model of P^+ by Proposition 6, $body(r') \subseteq M'$ implies $head(r') = a \in M'$. Because a is a positive literal, $a \in M$ holds. Hence, $T_{P^+}(M') \subseteq M$. Therefore, $M = T_{P^+}(M')$. □

Proposition 8. *Let M^+ be an interpretation of P+. If $T_{P^+}(M^+) = M^+ \cap B_P$ holds, then $M = M^+ \cap B_P$ is a supported model of P.*

Proof. Suppose $T_{P^+}(M^+) = M^+ \cap B_P$. Because $M^+ \cap B_P$ recovers the positive literals from M^+, for each $a \in (M^+ \cap B_P)$, there exists a rule $r \in P$ such that $head(r) = a$, $body^+(r) \subseteq (M^+ \cap B_P)$ and $body^-(r) \cap (M^+ \cap B_P) = \emptyset$. Thus, $M = M^+ \cap B_P$ is a supported model of P. □

4.3.2 Matrix Encoding of Logic Programs

In subsequent sections we shall use the following notations: matrices and vectors are represented as bold upper-case, $\mathbf{M}$, and lower-case letters,$\mathbf{v}$, respectively. A 1-vector with length N is represented by $\mathbf{1}_N$. The indices of the entries of matrices and vectors appear in the subscript, for example, $\mathbf{M}_{ij}$ refers to the element at i-th row and j-th column of a matrix $\mathbf{M}$ and $\mathbf{v}_i$ refers to the i-th element of a column vector $\mathbf{v}$. Let $\mathbf{M}_{i:}$ and $\mathbf{M}_{:j}$ denote the i-th row slice and j-th column slice of $\mathbf{M}$, respectively. We denote the horizontal concatenation of matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ as $[\mathbf{M}_1\ \mathbf{M}_2]$, and denote the vertical concatenation of column vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ as $[\mathbf{v}_1; \mathbf{v}_2]$.

Let P be a normal program with size $|B_P| = N$, P^+ its positive form and B_{P^+} the Herbrand base of P^+. Then we have $|B_{P^+}| = 2N$ since for every $b \in B_P$ we add its negated version $\overline{b}$. We encode atoms appearing in the bodies of the rules $\in P^+$ into a binary program matrix $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$.

Definition 5 (Program Matrix)**.** *Let P be a normal program with $|B_P| = N$ and P^+ its positive form with $|B_{P^+}| = 2N$. Then P^+ is represented by a matrix $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$ such that for each element $\mathbf{Q}_{ij} (1 \le i \le N, 1 \le j \le 2N)$ in $\mathbf{Q}$,*

- $\mathbf{Q}_{ij} = 1$ *if atom $a_j \in B_{P^+} (1 \le j \le 2N)$ appears in the body of the rule $r_i (1 \le i \le N)$;*
- $\mathbf{Q}_{ij} = 0$, *otherwise.*

The i-th row of $\mathbf{Q}$ corresponds to the atom a_i appearing in the head of the rule r_i, and the j-th column corresponds to the atom $a_j (1 \le j \le 2N)$ appearing in the body of the rules $r_i (1 \le i \le N)$. Atoms that do not appear in the head of any of the rules in P^+ are encoded as zero-only row vectors in $\mathbf{Q}$.

This definition is different from the previous works [51, 52], in that we do not explicitly include $\top$ and $\bot$ in the program matrix, and we do not use fractional values to encode long rules. In fact, our encoding method is similar to that of [55], except that

we do not use $(2N \times 2N)$ space for the program matrix since we do not encode rules with $\overline{b} \in B_{P^+}$ in the head.

Definition 6 (Interpretation Vector). *Let P be a definite program and $B_P = \{a_1, \ldots, a_N\}$. Then an interpretation $I \subseteq B_P$ is represented by a vector $\mathbf{v} = (\mathbf{v}_1, \ldots, \mathbf{v}_N)^\intercal \in \mathbb{Z}^N$ where each element $\mathbf{v}_i$ $(1 \leq i \leq N)$ represents the truth value of the proposition a_i such that $\mathbf{v}_i = 1$ if $a_i \in I$, otherwise $\mathbf{v}_i = 0$. We assume propositional variables share the common index such that $\mathbf{v}_i$ corresponds to a_i, and we write $\mathrm{var}(\mathbf{v}_i) = a_i$.*

Recall that the positive form P^+ of a normal program is a definite program, and all negated literals in the body are replaced by new atoms, e.g. in (4.4) $\neg b_{l+1}$ is replaced by $\overline{b}_{l+1}$. We now extend the definition of interpretation vectors to include relations between the positive and negative occurrences of atoms, to maintain whenever we have $b_1 \in I$, $\overline{b}_{l+1} \notin I$ and vice versa.

Definition 7 (Companion Vector). *Let $B^P_{P^+} \subseteq B_{P^+}$ denote the positive part of P, $B^N_{P^+} \subseteq B_{P^+}$ denote the negative part of P, with size $|B^P_{P^+}| = |B^N_{P^+}| = N$. Let $\mathbf{v}^P \in \mathbb{Z}^N$ be a vector representing truth assignments for $a_i \in B^P_{P^+}$ such that $\mathbf{v}^P_i = 1$ if $a_i \in I$ and $\mathbf{v}^P_i = 0$ otherwise. Define a companion vector $\mathbf{w} \in \mathbb{Z}^{2N}$ representing an interpretation $I^+ \subseteq B_{P^+}$ as follows: $\mathbf{w} = [\mathbf{v}^P; \mathbf{1}_N - \mathbf{v}^P]$.*

4.4 Gradient Descent For Computing Supported Models

4.4.1 Computing the T_P Operator in Vector Spaces

Sakama et al. [52] showed that the T_P operator can be computed in vector spaces using *θ-thresholding*. Here we modify θ-thresholding to accommodate our program encoding method as well as the differentiability requirement.

In previous works [58, 52], the information about the nature of the rules was also stored in the program matrix $\mathbf{Q}$ alongside the atom occurrences; conjunctive rules with $|body(r_i)| > 1$ had fractional values $\mathbf{Q}_{ij} = 1/|body(r_i)|$ and disjunctive bodies had integer values $\mathbf{Q}_{ij} = 1$. Instead, we only store the atom occurrence in $\mathbf{Q}$, and keep supplementary information in the parameter vector $\mathbf{t}$.

Definition 8 (Parameter Vector $\mathbf{t}$). *A set of parameters to the θ-thresholding is a column vector $\mathbf{t} \in \mathbb{Z}^N$ such that for each element $\mathbf{t}_i (1 \leq i \leq N)$ in $\mathbf{t}$,*

- $\mathbf{t}_i = |body(r_i)|$ *if the rule $r_i \in P^+$ is a conjunctive rule, e.g. (4.1), (4.4);*
- $\mathbf{t}_i = 1$ *if the rule $r_i \in P^+$ is a disjunctive rule e.g. (4.2);*
- $\mathbf{t}_i = 0$, *otherwise.*

Definition 9 (Parameterized θ-thresholding). *Let $\mathbf{w} \in \mathbb{Z}^{2N}$ be a companion vector representing $I^+ \subseteq B_{P^+}$. Given a parameter vector $\mathbf{t} \in \mathbb{Z}^N$, a program matrix $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$, and a vector $\mathbf{y} = \mathbf{Q}\mathbf{w}$ where $\mathbf{y} \in \mathbb{Z}^N$, we apply the thresholding function element-wise as follows:*

$$\theta_{\mathbf{t}}(\mathbf{y}_i) = \begin{cases} \min(\max(0, \mathbf{y}_i - (\mathbf{t}_i - 1)), 1) & (\mathbf{t}_i \geq 1) \\ 0 & (\mathbf{t}_i < 1) \end{cases} \tag{4.5}$$

This thresholding function resembles *hardtanh* which is an activation function developed for use in natural language processing [59]. In the original *hardtanh* function, the range of the linear region is $[-1, 1]$, but here we define the linear region between $[\mathbf{t}_i - 1, \mathbf{t}_i]$. This function is almost everywhere differentiable except at $\mathbf{y}_i = \mathbf{t}_i - 1$ and $\mathbf{y}_i = \mathbf{t}_i$. The special case $\mathbf{t}_i < 1$ in Equation (4.5) corresponds to the case $\mathbf{t}_i = 0$ where the head does not appear in the program P^+ and is assumed to be $false$.

Intuitively, for the head of a conjunctive rule to be *true* it is sufficient to check whether all literals in the body hold; otherwise the rule evaluates to $false$. Similarly, for a disjunctive rule, it is sufficient to check whether at least one of the literals in the body holds for the head to hold.

Proposition 9 (Thresholded T_P Operator). *Let P^+ be the positive form of a normal program P and $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$ its matrix representation. Suppose that $I^P \subseteq B^P_{P^+}$ is the positive part of an interpretation of P^+, and let $\mathbf{v}$ be its corresponding vector, i.e., $v_i = 1$ iff $a_i \in I^P$ and $v_i = 0$ otherwise, for $i = 1, ..., N$. Let $\mathbf{w} \in \mathbb{Z}^{2N}$ be the companion vector to $\mathbf{v}$. Then $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}] \in \mathbb{Z}^{2N}$ is the vector representing $J = T_P(I)$ satisfying the condition ($a \in J$ iff $\overline{a} \notin J$), iff $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{w})$.*

Proof. Consider $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{w})$. For $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_N)^\intercal$, by the definition of the thresholding function, $\mathbf{u}_k = 1 \, (1 \leq k \leq N)$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$ in $\mathbf{u}' = \mathbf{Q}\mathbf{w}$. Take a row slice

$\mathbf{Q}_{k:}$, then $\mathbf{u}'_k = \mathbf{Q}_{k:}\mathbf{w} = \mathbf{Q}_{k1}\mathbf{w}_1 + \cdots + \mathbf{Q}_{k2N}\mathbf{w}_{2N}$, and $\mathbf{u}_k = 1$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$. Both $\mathbf{Q}_{k:}$ and $\mathbf{w}$ are 0-1 vectors, then it follows that there are at least $\mathbf{t}_k$ elements where $\mathbf{Q}_{kj} = \mathbf{w}_j = 1\,(1 \leq j \leq 2N)$. The first N elements of $\mathbf{w}$ represent $a_i \in I^P \subseteq B^P_{P^+}$ if $\mathbf{w}_i = 1$, and if $a_i \in I^P$ then $\overline{a}_i \notin I^N \subseteq B^N_{P^+}$ which is maintained through the definition of the companion vector $\mathbf{w}$. 1) For a conjunctive rule $a_k \leftarrow a_1 \wedge \cdots \wedge a_m\,(1 \leq m \leq 2N)$, $\{a_1, \ldots, a_{2N}\} \in I$ implies $a_k \in T_P(I)$. 2) For an OR-rule $a_k \leftarrow a_1 \vee \cdots \vee a_m\,(1 \leq m \leq 2N)$, $\{a_1, \ldots, a_{2N}\} \subseteq I$ implies $a_k \in T_P(I)$. $a_m \in I$ is represented by $\mathbf{z}_m = 1\,(1 \leq m \leq 2N)$. Then by putting $J = \{\mathrm{var}(\mathbf{z}_m)|\mathbf{z}_m = 1\}$, $J = T_P(I)$ holds.

Consider $J = T_P(I)$. For $\mathbf{v} = (\mathbf{v}_1, \ldots, \mathbf{v}_N)^\intercal$ representing $I^P \subseteq B^P_{P^+}$, $\mathbf{w} = (\mathbf{v}_1, \ldots, \mathbf{v}_N, 1-\mathbf{v}_1, \ldots, 1-\mathbf{v}_N)^\intercal$ is a vector representing $I \subseteq B_{P^+}$ if we set $I = \{\mathrm{var}(\mathbf{w}_i)|\mathbf{w}_i = 1\}$. $\mathbf{u}' = \mathbf{Q}\mathbf{w}$ is a vector such that $\mathbf{u}'_k \geq \mathbf{t}_k\,(1 \leq k \leq N)$ iff $\mathrm{var}(\mathbf{u}'_k) \in T_P(I)$. Define $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_N)^\intercal$ such that $\mathbf{u}_k = 1\,(1 \leq k \leq N)$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$ in $\mathbf{Q}\mathbf{w}$, and $\mathbf{u}_k = 0$ otherwise. Define an interpretation $J \subseteq B_{P^+}$ such that it can be partitioned into subsets of positive and negative occurrences of atoms $(J^P \cup J^N) = J \subseteq B_{P^+}$. Since only positive atoms occur in the head, $\mathbf{u}$ represents a positive subset of interpretation $J^P \subseteq J \subseteq B_{P^+}$ by setting $J^P = \{\mathrm{var}(\mathbf{u}_i)|\mathbf{u}_i = 1\}\,(1 \leq i \leq N)$. If $a_i \in T_P(I)$ then $\mathbf{u}_i = 1$, and $\overline{a}_i \notin T_P(I)$ is represented by $1 - \mathbf{u}_i = 0$. Conversely, if $a_i \notin T_P(I)$ then $\mathbf{u}_i = 0$, and $1 - \mathbf{u}_i = 1$ represents $\overline{a}_i \in T_P(I)$. Thus $1 - \mathbf{u}$ represents $J^N \subseteq J \subseteq B_{P^+}$. $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$ is then a vector representing $J^P \cup J^N = J$ if we set $J = \{\mathrm{var}(\mathbf{z}_m)|\mathbf{z}_m = 1\}\,(1 \leq m \leq 2N)$. Thus $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$ represents $J = T_P(I)$ if $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{w})$.

□

Example 1. *Consider the following program:*

$$\begin{aligned} p &\leftarrow q \\ q &\leftarrow p \wedge r \\ r &\leftarrow \neg p \end{aligned} \tag{4.6}$$

This program has one supported (stable) model: $\{r\}$. *We have* $B_P = \{p, q, r\}$, $B_{P^+} =$

$\{p, q, r, \overline{p}, \overline{q}, \overline{r}\}$, *the matrix representation* $\mathbf{Q}$ *and parameter vector* $\mathbf{t}$ *are:*

$$\mathbf{Q} = \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{array}{c} \begin{array}{cccccc} p & q & r & \overline{p} & \overline{q} & \overline{r} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{t} = \begin{array}{c} p \\ q \\ r \end{array} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \tag{4.7}$$

Suppose an assignment $\mathbf{v}^{\{r\}} = (0\ 0\ 1)^\intercal$ *is given. The companion vector* $\mathbf{w}$ *is:*

$$\mathbf{w} = [\mathbf{v}^{\{r\}}; \mathbf{1}_3 - \mathbf{v}^{\{r\}}] = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}^\intercal \tag{4.8}$$

Compute the matrix multiplication product $\mathbf{Qw}$ *and apply the thresholding:*

$$\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Qw}) = \theta_{\mathbf{t}}(\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}^\intercal) = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^\intercal = \mathbf{v}^{\{r\}} \tag{4.9}$$

Let $\mathbf{z}$ *be a companion vector to* $\mathbf{u}$*, i.e.* $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$*, then we have*

$$\mathbf{z} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}^\intercal \tag{4.10}$$

Define $J = \{\mathrm{var}(\mathbf{z}_m) | \mathbf{z}_m = 1\}$*, then we have* $J = \{r, \overline{p}, \overline{q}\}$*, and* $J \cap B_P = \{r\}$*.*

Proposition 10 (Supported Model Computation with Thresholded T_P)**.** *Let* $\mathbf{v} \in \mathbb{Z}^N$ *be a 0-1 vector representing a subset of interpretation* $I^P \subseteq I \subseteq B_{P^+}$*, and* $\mathbf{z} = [\mathbf{v}; \mathbf{1}_N - \mathbf{v}]$ *be its companion vector representing* $I \subseteq B_{P^+}$ *satisfying* ($a \in I$ *iff* $\overline{a} \notin I$)*. Given a program matrix* $\mathbf{Q}$ *representing a program* P^+ *and a thresholding function* $\theta_{\mathbf{t}}$ *parameterized by a vector* $\mathbf{t}$*, the fixed points of* P^+ *are represented by 0-1 binary vectors* $\mathbf{z}^{FP} = [\mathbf{v}^{FP}; \mathbf{1}_N - \mathbf{v}^{FP}] \in \mathbb{Z}^{2N}$ *where* $\mathbf{v}^{FP} = \theta_{\mathbf{t}}(\mathbf{Qz}^{FP})$*. Then* $\mathbf{z}^{FP}$ *are vectors representing models* M^+ *of* P^+ *satisfying* ($a \in M^+$ *iff* $\overline{a} \notin M^+$) *iff* $\theta_{\mathbf{t}}(\mathbf{Qz}^{FP}) = \mathbf{v}^{FP}$*. When such 0-1 binary vector* $\mathbf{z}^{FP}$ *exists,* $M^+ \cap B_P = M$ *is a supported model of* P*.*

Proof. Let $I \subseteq B_{P^+}$ be a model of P^+, represented by $\mathbf{z}^{FP}$. Consider two cases (i) $T_P(I) = I$ and (ii) $\mathbf{v}^{FP} = \theta_{\mathbf{t}}(\mathbf{Qz}^{FP})$. In both cases, by Propositions 7, 8 and 9, if a supported model of P exists, the results hold.

□

4.4.2 Loss Function for Computing Supported Models

By the fixed point definition of supported models, a supported model M satisfies $\mathbf{v}^{M^P} = \theta_{\mathbf{t}}(\mathbf{Q}[\mathbf{v}^{M^P}; \mathbf{1}_N - \mathbf{v}^{M^P}])$. We now use this definition to design a loss function which can be minimized using gradient descent. Gradient descent is a method for minimizing an objective function (loss function), by updating the parameters in the opposite direction of the gradient of the objective function with respect to the parameters. The size of the update is determined by the gradient and the step size α.

We define a vector $\mathbf{f} \in \mathbb{Z}^N$ which stores information about occurrences of facts in the program P^+. This vector will be used later during the minimization step to ensure that facts are not forgotten.

Definition 10 (Fact Vector $\mathbf{f}$)**.** *The set of facts in the program P^+ is represented by a column vector $\mathbf{f} \in \mathbb{Z}^N$, such that for each element $\mathbf{f}_i (1 \leq i \leq N)$,*

- $\mathbf{f}_i = 1$ *if the rule r_i is a fact: $a \leftarrow$*
- $\mathbf{f}_i = 0$ *otherwise.*

Definition 11 (Loss Function)**.** *Given a program matrix $\mathbf{Q}$, a candidate vector $\mathbf{x}$, thresholding function $\theta_{\mathbf{t}}$, and constants λ_1 and λ_2, define the loss function as follows:*

$$L(\mathbf{x}) = \frac{1}{2}\Big(\|\theta_{\mathbf{t}}(\mathbf{Q}[\mathbf{x}; \mathbf{1}_N - \mathbf{x}]) - \mathbf{x}\|_{\mathrm{F}}^2 + \lambda_1 \|\mathbf{x} \odot (\mathbf{x} - \mathbf{1}_N)\|_{\mathrm{F}}^2 + \lambda_2 \|\mathbf{f} - (\mathbf{x} \odot \mathbf{f})\|_{\mathrm{F}}^2 \Big) \quad (4.11)$$

where $\|\mathbf{x}\|_{\mathrm{F}}$ denotes the Frobenius norm and $\odot$ element-wise product.

The first term is derived directly from the fixed point definition of supported models, and should be 0 if $\mathbf{x}$ is a supported model of P^+. The second term, which resembles a regularization term often used in the machine learning literature, is added to penalize candidate vectors $\mathbf{x}$ that contain fractional values, and is 0 if and only if $\mathbf{x}$ is a 0-1 vector. The third term will be 0 if and only if the facts are preserved, and will be positive non-zero if any part of the assignment is lost, i.e. by assigning 0 ($false$) to a fact where $\mathbf{f}_i = 1$.

We introduce submatrices of $\mathbf{Q}$, $\mathbf{Q}_p \in \mathbb{Z}^{N \times N}$ and $\mathbf{Q}_n \in \mathbb{Z}^{N \times N}$ that correspond to the positive bodies and negative bodies of the matrix, respectively, such that $\mathbf{Q} = [\mathbf{Q}_p\ \mathbf{Q}_n]$ (horizontal concatenation of submatrices).

Definition 12 (Gradient of the Loss Function). *The gradient of the loss function with respect to* $\mathbf{x}$ *is given by:*

$$\frac{\partial L(\mathbf{x})}{\partial \mathbf{x}} = \left((\mathbf{Q}_p - \mathbf{Q}_n)^T \cdot \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}}) \odot \frac{\partial \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}})}{\partial \mathbf{x}}\right) - \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}} - \mathbf{x}) + \lambda_1(\mathbf{x} \odot (\mathbf{1}_N - \mathbf{x}) \odot (\mathbf{1}_N - 2\mathbf{x})) + \lambda_2(\mathbf{x} \odot \mathbf{f} - \mathbf{f}) \tag{4.12}$$

where $\mathbf{z}_{\mathbf{x}} \in \mathbb{R}^{2N} = [\mathbf{x}; \mathbf{1}_N - \mathbf{x}]$ *and*

$$\frac{\partial \theta_{\mathbf{t}}(\mathbf{w}_i)}{\partial \mathbf{x}_i} = \begin{cases} 1 & \text{if } (\mathbf{t}_i \geq 1) \text{ and } (\mathbf{t}_i - 1) \leq \mathbf{w}_i \leq \mathbf{t}_i \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

We can update $\mathbf{x}$ iteratively using, for example, gradient descent or quasi-Newton's method, to reduce the loss to zero. Here we show an example of update rule for gradient descent. Let α be the step size, then the gradient descent update rule is given by:

$$\mathbf{x}_{\text{new}} \leftarrow \mathbf{x} - \alpha \frac{\partial L(\mathbf{x})}{\partial \mathbf{x}} \tag{4.14}$$

Using this update rule we can design an algorithm to find supported models, as shown in Algorithm 4.1. Moreover, this formulation allows us to use other gradient update methods like Newton update [56] or more advanced optimizers like Adam [60], as we show later in the experiment section.

The convergence characteristics of the gradient descent algorithm are well-known. Assuming at least one 0-1 vector representing a supported model exists for $\mathbf{Q}$, all we require for Algorithm 4.1 to converge to the supported model is that the initial vector $\mathbf{x}$ to be in the region surrounding the supported model where the slope points towards the model. When there are multiple supported models, we expect the algorithm to converge to different models depending on the choice of initial vector. However, it is often not known *apriori* which particular values or regions of $\mathbf{x}$ lead to which models. Thus, we implement a uniform initialization strategy, where the initial values are drawn from the uniform distribution $\mathcal{U}(0, 1)$.

Depending on the program, an optimal 0-1 vector interpretation may not exist, so we limit the number of iterations to max_iter before assuming non-convergence. With gradient descent, it is often time-consuming to reduce the loss function completely to

zero. We therefore implement a "peeking at a solution" heuristic, similar to the one presented in [56], where while updating $\mathbf{x}$ we round $\mathbf{x}$ to a 0-1 vector to see whether the resulting $\mathbf{x}_r$ is a solution (Lines 6-8). The output is sensitive to the choice of initial vector $\mathbf{x}$, and a poor choice may result in non-convergence to optimal solutions. We alleviate this dependency on the initial vector by introducing the max_retry parameter and changing the initial vector on each try. This algorithm declares failure to find any supported models (returns FALSE, Line 12) when both max_retry and max_iter are exhausted.

Algorithm 4.1 Gradient descent search of supported models

Input: Program matrix $\mathbf{Q}$, Thresholding parameter $\mathbf{t}$, max_retry ≥ 1, max_iter ≥ 1, $\epsilon > 0$, step size $\alpha > 0$, $\lambda_1 > 0$, $\lambda_2 > 0$
Output: Supported model $\mathbf{x}$ or FALSE

1: **for** n_try $\leftarrow 1$ **to** max_retry **do**
2: $\quad\mathbf{x} \leftarrow$ vector sampled from $\mathcal{U}(0, 1)$
3: $\quad$**for** n_iter $\leftarrow 1$ **to** max_iter **do**
4: $\quad\quad\mathbf{x}_r \leftarrow \text{round}(\mathbf{x})$ ▷ Rounding heuristic
5: $\quad\quad\text{loss} \leftarrow L(\mathbf{x}_r)$ ▷ Loss function, see Def. (11)
6: $\quad\quad$**if** (loss $\leq \epsilon$) **then**
7: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x}_r$
8: $\quad\quad\quad$**return** $\mathbf{x}$
9: $\quad\quad$**else**
10: $\quad\quad\quad\text{gradient} \leftarrow \frac{\partial L(\mathbf{x})}{\partial \mathbf{x}}$ ▷ Gradient, see Def. (12)
11: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} - \alpha \cdot \text{gradient}$ ▷ Gradient update
12: **return** FALSE

4.4.3 Restart Methods

While we use random sampling from the uniform distribution by default for the initial assignments (Line 2 in Algorithm 4.1), we can modify the assignment method to potentially improve the performance of our method. Formally, the restart method in this section concerns the initial assignment of the second try onwards, and not the initial assignment of the first try (n_try ≥ 2 in Line 2 of Algorithm 4.1). For effectively setting the initial assignments on the first try, it requires some form of prior knowledge on the problems, for example, a machine learning model that was trained on a set of random instances to predict the assignment (interpretations) of atoms. In this chapter,

unless otherwise noted, the restart method used is 'random sampling' by default in all experiments.

Random Sampling

At each try, we set the initial assignment as follows:

$$\mathbf{x}_{init} = \mathbf{u} \quad \text{where } \mathbf{u} \sim \mathcal{U}(0, 1)$$

Notably, this essentially discards the results of the last failed attempt, and starts again from a new set of candidate assignments.

Noise addition

At each try, we add perturbations sampled from the uniform distribution to the last iterate:

$$\mathbf{x}_{init} = \mathbf{x}_{last} + \mathbf{n} \quad \text{where } \mathbf{n} \sim \mathcal{U}(-0.5, 0.5)$$

The intuition behind this update is that we assume the last failed iterate is at a local minimum, and we aim to escape the local minimum by adding random noise to the last iterate.

Noisy rounding

At each try, we add perturbations sampled from the uniform distribution to the *rounded* last iterate:

$$\mathbf{x}_{init} = \text{round}(\mathbf{x}_{last}) + \mathbf{n} \quad \text{where } \mathbf{n} \sim \mathcal{U}(-0.5, 0.5)$$

This is essentially a combination of the 'peeking at the solution' heuristic and noise addition, where we aim simultaneously to escape the local minimum and move closer to an exact 0-1 interpretation vector.

Delta method

At each try, we apply margin thresholding and add a noise vector sampled from the uniform distribution. Instead of rounding to 0-1 vector, we apply a stricter version of

rounding with a constant δ.

$$\mathbf{x}_{init} = \text{round}_\delta(\mathbf{x}_{last}) \text{ where for each element } \mathbf{x}_i \in \mathbf{x}_{last}$$
$$\mathbf{x}_i = \begin{cases} 0 \text{ if } -\delta \leq \mathbf{x}_i \leq \delta \\ 1 \text{ if } (1-\delta) \leq \mathbf{x}_i \leq (1+\delta) \\ \mathbf{x}_i \text{ otherwise} \end{cases}$$

Additionally, if the resulting vector from $\text{round}_\delta(\mathbf{x}_{last})$ only contains 0 or 1, then we add a noise vector sampled from the uniform distribution, i.e.,

$$\mathbf{x}_{init} = \text{round}_\delta(\mathbf{x}_{last}) + \mathbf{n} \quad \text{where } \mathbf{n} \sim \mathcal{U}(-0.5, 0.5)$$

For the experiments, the value of δ is fixed at 0.1. The intuition behind this restart method is that, when the last value is 'close enough' to a 0-1 value, we take it as the next initial vector, and if the last value is outside the range, we update it in the next try.

Tabu list

This is a variation of the Tabu search, a popular metaheuristic used in combinatorial optimizations. We do not use the 'aspiration criteria', since the gradient-based search can perform local search on its own. The pseudocode for the tabu list restart method is shown in Algorithm 4.2. The intent of this method is to avoid getting stuck in the same

Algorithm 4.2 Tabu list restart method

Input: Last iterate $\mathbf{x}_{last}$
Output: $\mathbf{x}_{init}$

```
tabu_list = FIFO list of fixed length L
flip_index = randomly pick an index (not in tabu_list)
if x_last[flip_index] > 0.5 then
    x_init[flip_index] = 0
else
    x_init[flip_index] = 1
return x_init
```

local minimum again by keeping a record of past flips, so that same flip cannot be allowed in $|tabu\_list|$ retries.

4.5 Experiments

All experiments in this section were performed on a desktop machine with the following specifications: Python 3.7, Intel Core i9-9900K and 64GB RAM.

4.5.1 *N*-negative loops

Aspis et al. [51] encode the program reduct into a matrix and employ the Newton's method for root finding to find fixed points of the program. Their matrix encoding assumes the MD condition, whereas ours assumes the SD condition. The gradient is calculated by a Jacobian matrix, and the thresholding operation is carried out with a parameterized sigmoid. They present two types of sampling methods for setting the initial vector; *uniform sampling*, similarly to our method, where the values are sampled uniformly from $[0, 1]$, and *semantic sampling*[1], where the values are sampled uniformly from $[0, \gamma^{\bot}] \cup [\gamma^{\top}, 1]$.

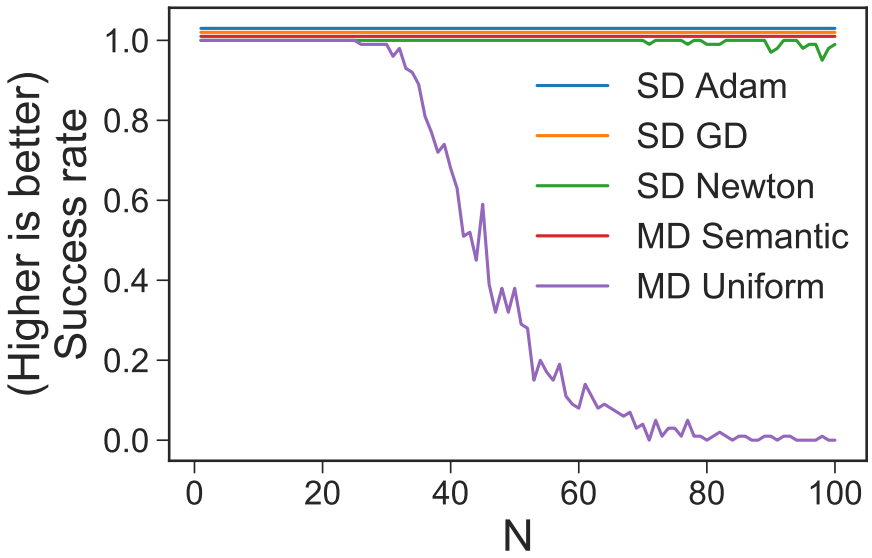
Firstly, we consider the "N-negative loops" programs, which involves programs in the following form: for $1 \leq i \leq N$,

$$\begin{aligned} p_i &\leftarrow \text{not } q_i \\ q_i &\leftarrow \text{not } p_i \end{aligned} \tag{4.15}$$

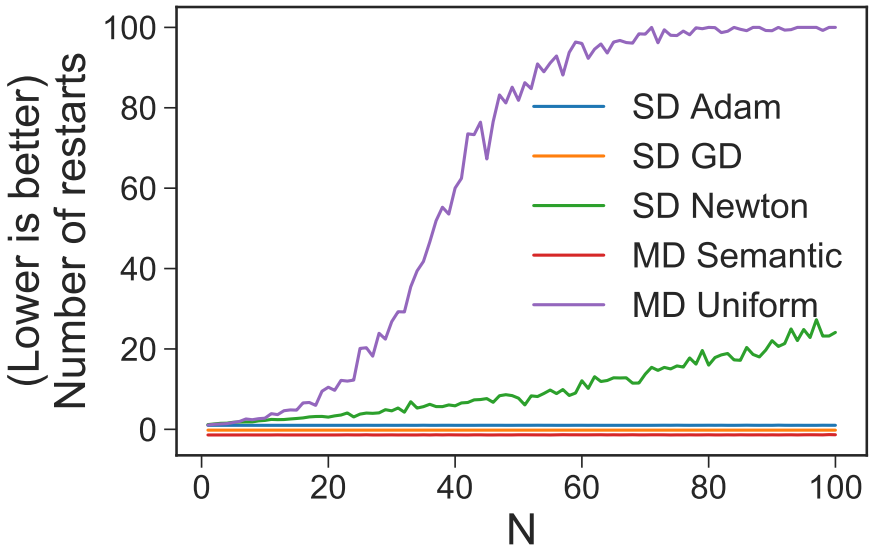
For our algorithm, we use the following parameters: max_iter $= 10^3$, $\epsilon = 10^{-4}$, $\lambda_1 = \lambda_2 = 1$, $\alpha = 10^{-1}$. For comparison, we also implemented Aspis et al.'s algorithm, and we used the following settings: max_iter $= 10^3$, $\epsilon = 10^{-4}$. Both algorithms were allowed to restart from a new random vector up to 100 times, in case the iteration fails to find a model. We generated programs of the form Program (4.15) with N up to 100, and then applied the algorithms on each program 10 times. We measured the rate of success of converging to supported models, and the number of restarts attempted by the algorithms (Figure 4.1a and Figure 4.1b).

From Figure 4.1a, one can observe that our method, except for the Newton update at around $N = 98$, could find the correct supported models regardless of the gradient

[1] $\gamma^{\bot}$ is an upper bound on false values that variables can take, and $\gamma^{\top}$ is a lower bound on true values. $\gamma = \frac{n}{n+1}$ where n is the length of the longest positive part in the rules, and τ was estimated as described.[51]



(a) Success rate



(b) Number of restarts

Figure 4.1: N-negative loops. Success rate of computing supported models. SD: Singly-Defined program, MD: Multiple Definition.

update method. The gradient descent and Adam updates can solve this task with the least number of restarts, and in fact, they found the models on their first attempts (Figure 4.1b). On the other hand, we see a gradual increase in the number of restarts required for the Newton update method, and the MD method requires more than 100 restarts past $N = 40$ to solve.

The design of the loss function (Section 4.4.2) also contributes to the high success rate of our method. The second term results in the loss function having non-zero values at local minima where the interpretation vector is non-binary, and drives the optimizers away from local minima. The root finding method described in [51] without this penalty term is prone to the presence of local minima, as shown by the low success rates.

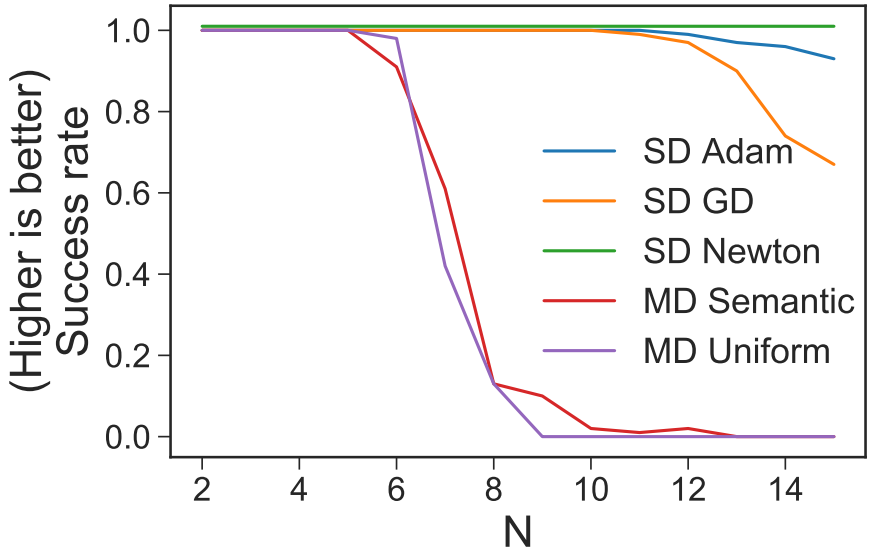
4.5.2 Choose 1 out of N

Secondly, we consider the "choose 1 out of N" task, where the task is to choose exactly 1 out of N options. The programs in this class have the following form:

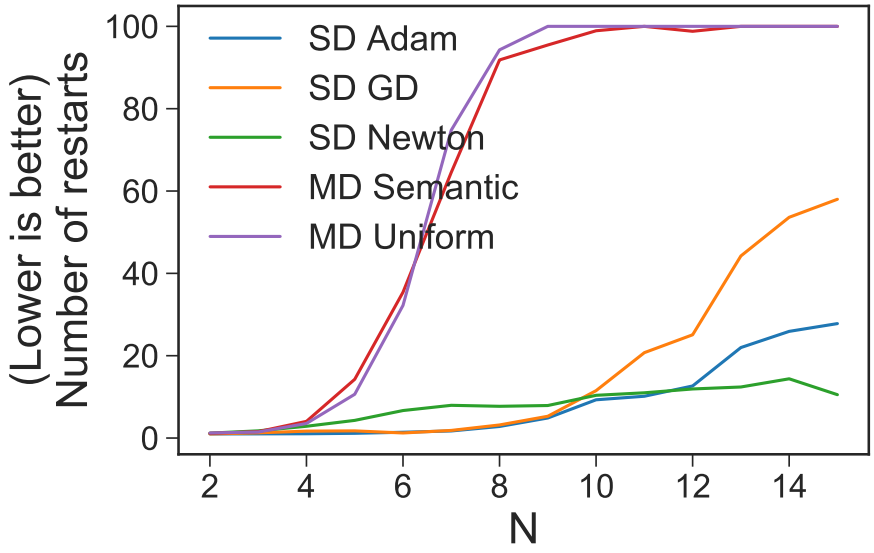
$$\begin{aligned} p_1 &\leftarrow \text{not}\, p_2,\ \text{not}\, p_3,\ ...,\ \text{not}\, p_N \\ p_2 &\leftarrow \text{not}\, p_1,\ \text{not}\, p_3,\ ...,\ \text{not}\, p_N \\ &\vdots \\ p_N &\leftarrow \text{not}\, p_1,\ \text{not}\, p_2,\ ...,\ \text{not}\, p_{N-1} \end{aligned} \tag{4.16}$$

We generated programs for N between 2 and 14, and applied the algorithms using the same parameters as the "N-negative loops" task, and repeated the process for 10 times for each N.

In contrast to the previous case, the Newton update turned out to be the most stable, followed closely by Adam and GD (Figure 4.2). Moreover, for gradient descent, we observe a steeper increase in the number of restarts past $N = 10$ compared to Adam (Figure 4.2b). This suggests that adaptive gradient methods, that can change the step size on the fly, may be better suited for more complex programs.

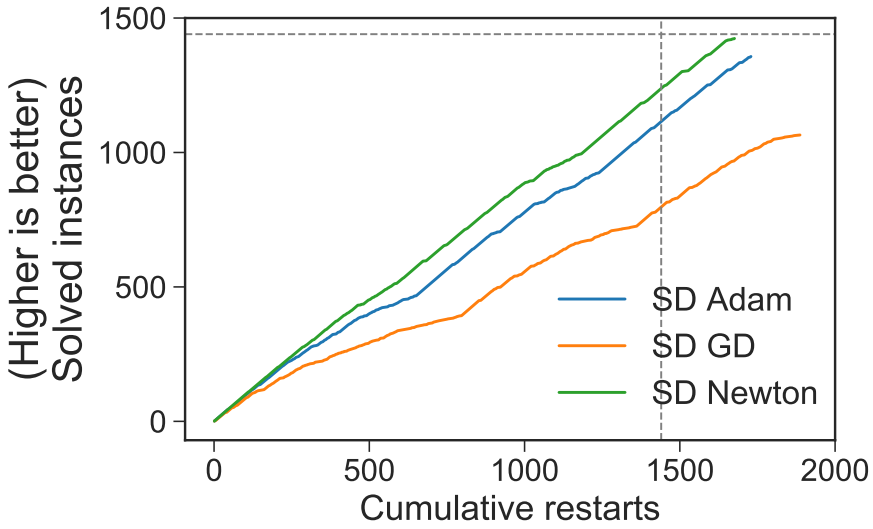


(a) Success rate

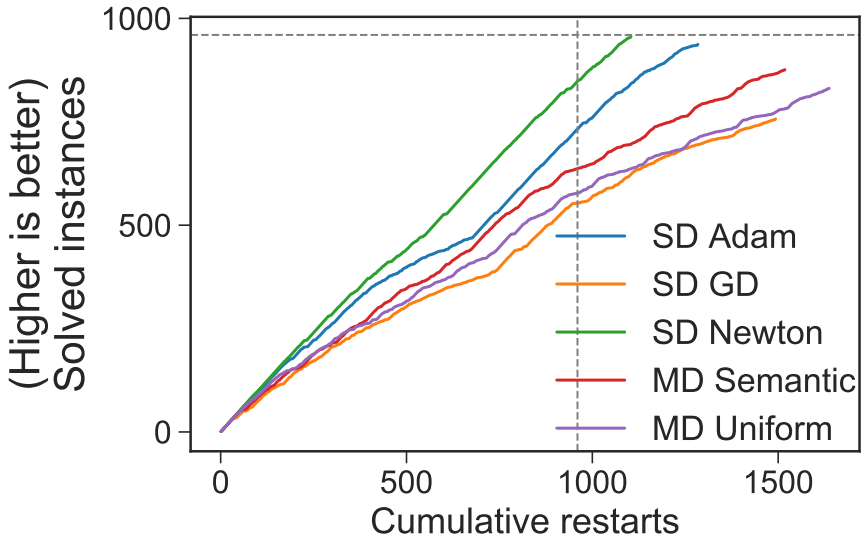


(b) Number of restarts

Figure 4.2: Choose 1 out of N. Success rate of computing supported models. SD: Singly-Defined program, MD: Multiple Definition.



(a) Full instances (1,440)



(b) Restricted instances (960)

Figure 4.3: Randomly generated programs. SD: Singly-Defined program, MD: Multiple Definition.

4.5.3 Random Programs

To evaluate our approach on a wide range of programs, we generated random programs by changing (a) number of atoms $[10, 50, 100]$ (b) number of rules, as a multiplier on the number of atoms $[1, 1.2, 1.4, ..., 2, 2.5, 3]$ and (c) % chance of literals in the body being negative $[0, 10, 30, ..., 90]$, and generated 10 instances for each set of parameters (1,440 instances). Note that we did not fix the size of the body, and the size was sampled from a uniform distribution $\mathcal{U}(0, |B_P| - 1)$ where $|B_P|$ is the number of unique atoms in the program. During the evaluation, it became apparent that running the MD methods on programs containing long rules with negation was very time-consuming. Thus, we created a second set of randomly generated programs, where the number of atoms was restricted to $[10, 15]$ (960 instances). We applied the algorithms once to each of the instances with max_try = 10 and max_iter = 100, and recorded success when the algorithm reached respective convergence criteria.

In Figure 4.3, we plot the number of solved instances vs. cumulative number of restarts. The dashed lines indicate the 'perfect score', which is achievable if the algorithm could find the model successfully on the first try for all instances. Thus, the closer an algorithm is to this point, the more efficient it is in finding the correct models. Overall, we see that the Newton update performs the best, followed by Adam. On smaller datasets, we found that gradient descent was less efficient than the MD method in terms of number of restarts; however, we may be able to improve the result by simply increasing the step size.

Run time comparison

As mentioned in the previous section, we found that the run time of the MD methods on programs containing negations was longer than the SD method. We found two potential causes for this observation: (a) the evaluation of the program reduct in itself is time-consuming, and it is exacerbated by the fact that the reduct has to be calculated multiple times in each iteration step (b) the use of external solver in each iteration. (b) can be fast if the initial configuration was sufficiently close to the potential solution; however, it can also be slow when the solver has to run to its max-iteration setting without actually finding the answer. In general, the larger the program is, it is more likely that the initial configuration is far from the potential solution.

As for (a), this stems from the matrix representation of the program using the Multiple Definition condition, and because the interpretation vector is updated at each iteration, the program reduct also has to be updated at each iteration. For brevity, in the following example, we shall omit rules that are not directly relevant.

Example 2. *Consider the following program:*

$$
\begin{aligned}
p &\leftarrow q \\
p &\leftarrow r \\
p &\leftarrow s \wedge t \wedge u \\
&\ldots(omitted)
\end{aligned}
$$

This program satisfies the MD condition, since there is only 1 rule whose head is p and the length of the body is greater than 1. On the other hand, this program does not satisfy the SD condition, so it is translated into:

$$
\begin{aligned}
p &\leftarrow p_1 \vee p_2 \vee p_3 \\
p_1 &\leftarrow q \\
p_2 &\leftarrow r \\
p_3 &\leftarrow s \wedge t \wedge u \\
&\ldots(omitted)
\end{aligned}
$$

The matrix representation of the MD program, $\mathbf{P}_{MD}$, according to Sakama et al.'s encoding [52] on which Aspis et al.'s work [51] is based on, is:

$$
\mathbf{P}_{MD} = \begin{array}{c} \\ p \end{array} \begin{array}{c} \begin{array}{cccccc} p & q & r & s & t & u \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1/3 & 1/3 & 1/3 \\ & & & \ldots & & \end{pmatrix} \end{array}
$$

On the other hand, in our encoding of the SD program (negative literals omitted),

$$\mathbf{P}_{SD} = \begin{array}{c} \\ p \\ p_1 \\ p_2 \\ p_3 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccc} p & p_1 & p_2 & p_3 & q & r & s & t & u \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ & & & & \cdots & & & & \end{pmatrix} \end{array}$$

Note that, aside from the obvious use of fractional values to represent long rules, the MD matrix is more densely packed compared to the SD matrix, and the SD representation is more 'unraveled'. In Aspis et al. [51], the matrix representation of the program reduct is defined as a collection of summation operations over each head atom, resulting in a matrix that is in the form of $\mathbf{P}_{MD}$ with fractional values. Fractional values may arise from the negative literals in the body in their definition, but we have omitted it for simplicity.

In our case (SD representation), for implementing the immediate consequence operator (T_P) in a continuous domain, we require only a matrix-vector multiplication with thresholding. On the other hand, in Aspis et al.'s case, the aforementioned summation over head atoms is required in addition to the matrix-vector multiplication and thresholding. We shall now demonstrate the effect of this difference in terms of run times of both algorithms. For this experiment, we used the restricted set (960 instances) of the randomly generated programs (Figure 4.4).

Figure 4.4 shows the number of solved instances against (log) run time in seconds. Regardless of the initialization method, the MD method overall requires orders of magnitude more time to solve an instance compared to the SD method. This demonstrates another advantage of using our gradient-based computation method instead of root-finding-based algorithm presented in Aspis et al. [51]: our algorithm can more effectively take advantage of the matrix processing routines and is much faster.

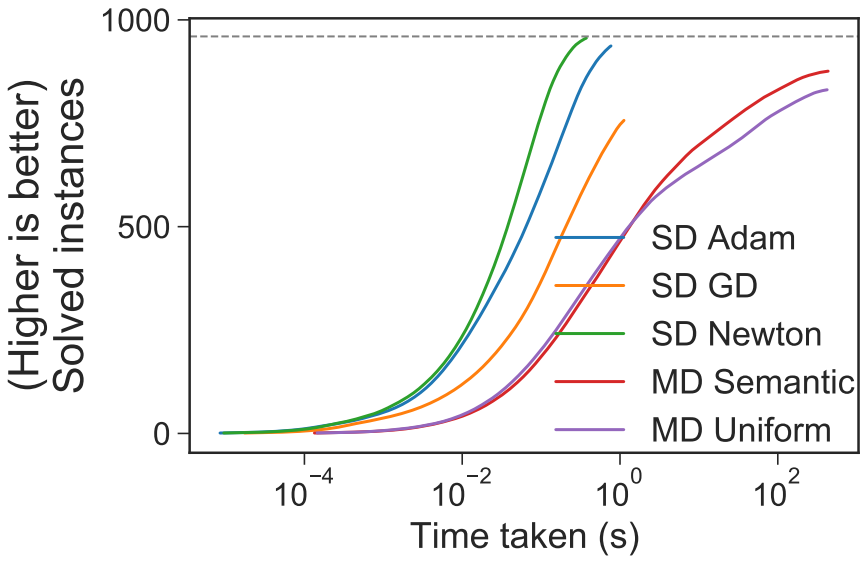
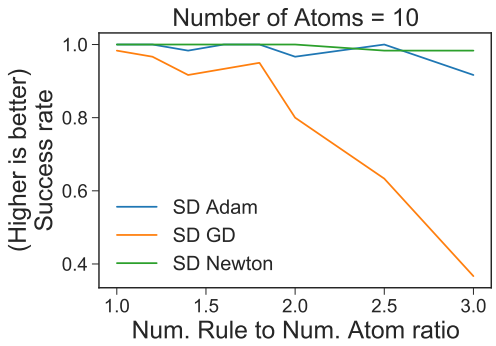


Figure 4.4: Run time on Randomly generated programs. SD: Singly-Defined program, MD: Multiple Definition.

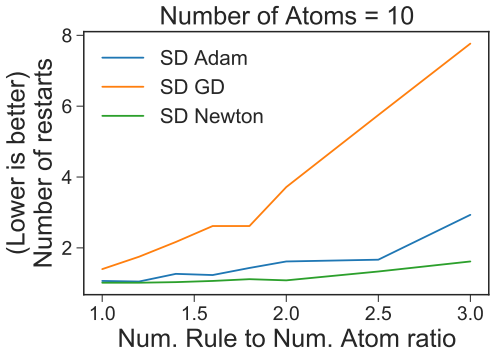
Effects of same-head rules and negation

The random programs in this section were generated with varying number of atoms, rules and % chance of negation in the body. In this subsection, we shall study the effects of adding same-head rules and having more negative literals in the body. Recall that when we translate normal logic programs into SD programs, we replace the same-head rules with sets of new rules, which includes rules with replaced heads and OR-rules. Thus, varying the rules-to-atoms ratio allows us to study the effect on performance of adding same-head rules to the program. As for the chance of negation, recall from the "choose 1 out of N" task that the performance of differentiable logic programming methods deteriorates quite rapidly as the program size increases, when the chance of negation is at 100% (all literals in the body are negative). The random programs include variations from 0% negation (definite program, no negation in the body) to 90% negation (close to "choose 1 out of N" situation).

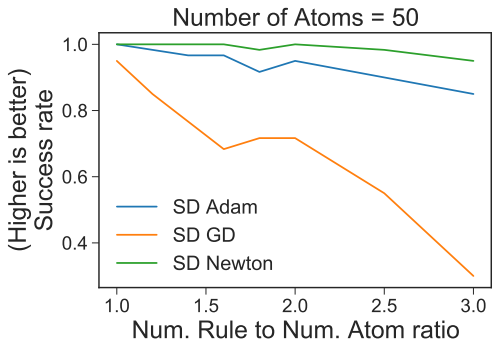
Figure 4.5 shows the success rates and number of restarts at different settings. The ratio of number of rules to atoms can be interpreted as follows: when the ratio is 1, there are no rules with duplicate head atoms, so there will be no OR-rules in the resulting SD program. When the ratio is 2, there are twice as many rules as atoms, so



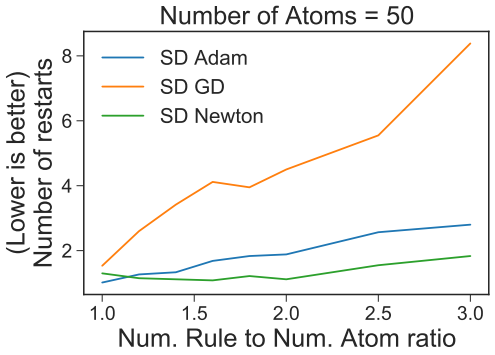
(a) Success rate of finding supported models. Number of atoms = 10



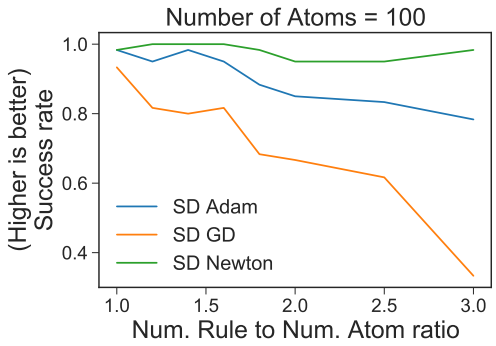
(b) Number of restarts (successful runs only). Number of atoms = 10



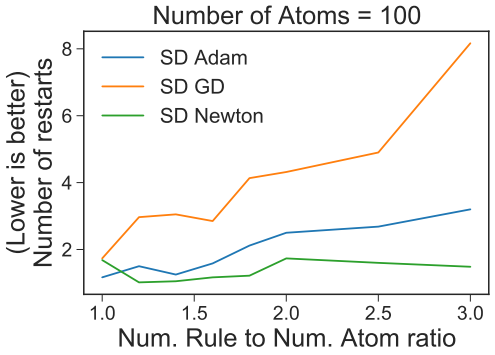
(c) Success rate of finding supported models. Number of atoms = 50



(d) Number of restarts (successful runs only). Number of atoms = 50



(e) Success rate of finding supported models. Number of atoms = 100



(f) Number of restarts (successful runs only). Number of atoms = 100

Figure 4.5: Randomly generated programs. Effect of adding same-head rules.

in the resulting SD program, there can be multiple OR-rules.

As for the gradient update rules, the trend remains the same: the Newton update (adaptive gradient) performed the best overall, followed by Adam, and the vanilla gradient descent performed the worst. For gradient descent, the success rate of finding supported models drops quite rapidly as the rules to atom ratio increases. For other update rules, the drop is not as pronounced as that of gradient descent, but the effect is still noticeable, as we observe a consistent drop in success rates when the ratio reaches 3.

The plots for the number of restarts required to find supported models successfully show a similar trend. As the ratio increases, the number of restarts increases, and this holds for all update methods. As these plots show, the number of same-head rules in the original program has a negative impact on the performance of our differentiable method for computing supported models.

Figure 4.6 shows the success rates and number of restarts at different settings of % chance of negation in body literals. Note that this percentage is not to be understood as the ratio of negated literals in the body, rather, it is the % chance of each literal being negated when generating the programs. The overall trend is similar to the same-head case; adding more negated literals to the body negatively impacts the performance of our method.

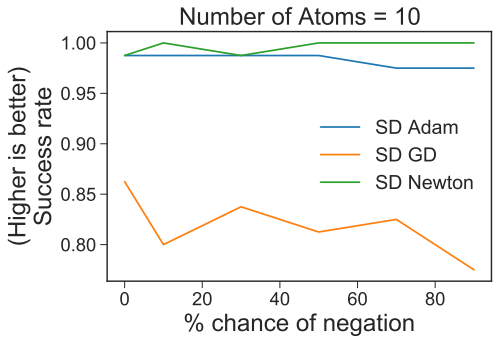
The success rate starts dropping rapidly past 50% in N=50 and N=100 for gradient descent update (Figures 4.6c, 4.6e), but this drop is delayed until 70% for Newton update and Adam. For small programs (N=10), this drop was not observed (note that gradient descent achieves around 80% success rates overall in Figure 4.6a). The number of restarts required for the algorithm to find supported models increases rapidly past 50% for N=50 and N=100 (Figure 4.6d, 4.6f).

Overall, these experiments demonstrated that there are potentially two classes of programs which can be considered to be difficult for our method to solve.

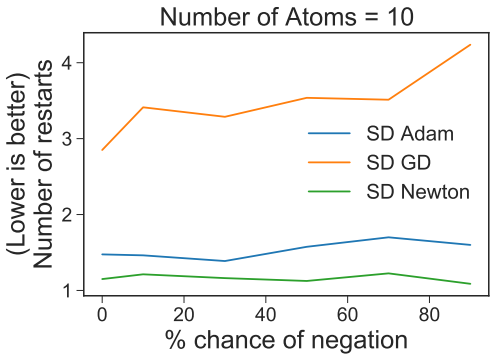
1. Programs with more than one set of same-head rules, which results in multiple OR-rules in the translated SD programs.

2. Programs with high number of negated literals in the bodies of the rules.

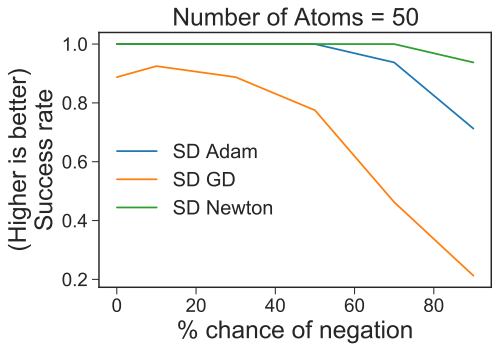
Further work on this algorithm could consider improvements in the aforementioned



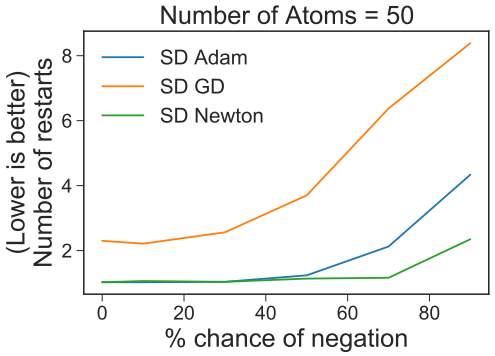
(a) Success rate of finding supported models. Number of atoms = 10



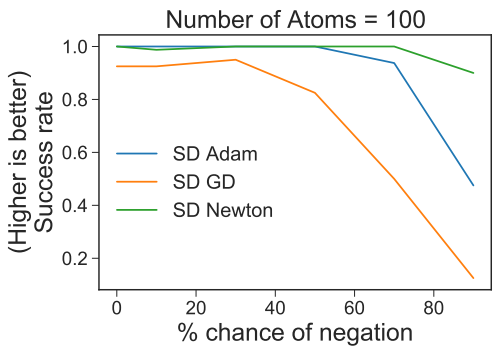
(b) Number of restarts (successful runs only). Number of atoms = 10



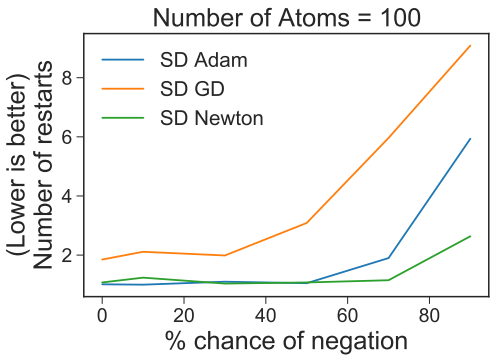
(c) Success rate of finding supported models. Number of atoms = 50



(d) Number of restarts (successful runs only). Number of atoms = 50



(e) Success rate of finding supported models. Number of atoms = 100



(f) Number of restarts (successful runs only). Number of atoms = 100

Figure 4.6: Randomly generated programs. Effect of negation.

classes of programs. These improvements might update the thresholding operation, the restart methods and potentially the loss function itself.

4.5.4 GPU implementation

Because our method is based around vectors and differentiable, it is rather trivial to re-implement our method for GPU to leverage parallel computational power of modern GPUs. In this subsection, we study the performance of the GPU implementation compared against purely CPU-based versions. It should be noted that the current implementation is preliminary and therefore not fully optimized yet, and performance is likely to be improved significantly by optimizing for GPU further.

Implementation details

We compare the following implementations: (a) CPU (b) CPU-JIT and (c) GPU. All implementations are primarily based on the Python language, while (a) is based purely on the NumPy package, (b) is based on the NumPy package but accelerated by a JIT compiler offered by the Numba package, and (c) is based on the PyTorch package, where all matrix operations are offloaded to the GPU (CUDA). We expect (b) to be faster than (a), but we are interested in the relative performance of (c) because the speed of (c) is highly implementation- and problem- dependent. In general, GPUs tend to be faster than CPU when it comes to floating-point matrix multiplication tasks, but using GPUs incur overhead of transferring data from CPU-side to GPU. For smaller matrices, therefore, we expect the CPU versions to outperform the naive GPU implementation.

Experimental details

The programs used in this subsection are a superset of the Random Programs. We generated random programs by changing (a) number of atoms $[10, 50, 100, 150, 200]$ (b) number of rules, as a multiplier on the number of atoms $[1, 1.2, 1.4, ..., 2, 2.5, 3]$ and (c) % chance of literals in the body being negative $[0, 10, 30, ..., 90]$, and generated 10 instances for each set of parameters (2,400 instances). We applied the algorithms 10 times to each of the instances with max_try = 100 and max_iter = 100, and recorded success when the algorithm reached the convergence criteria.

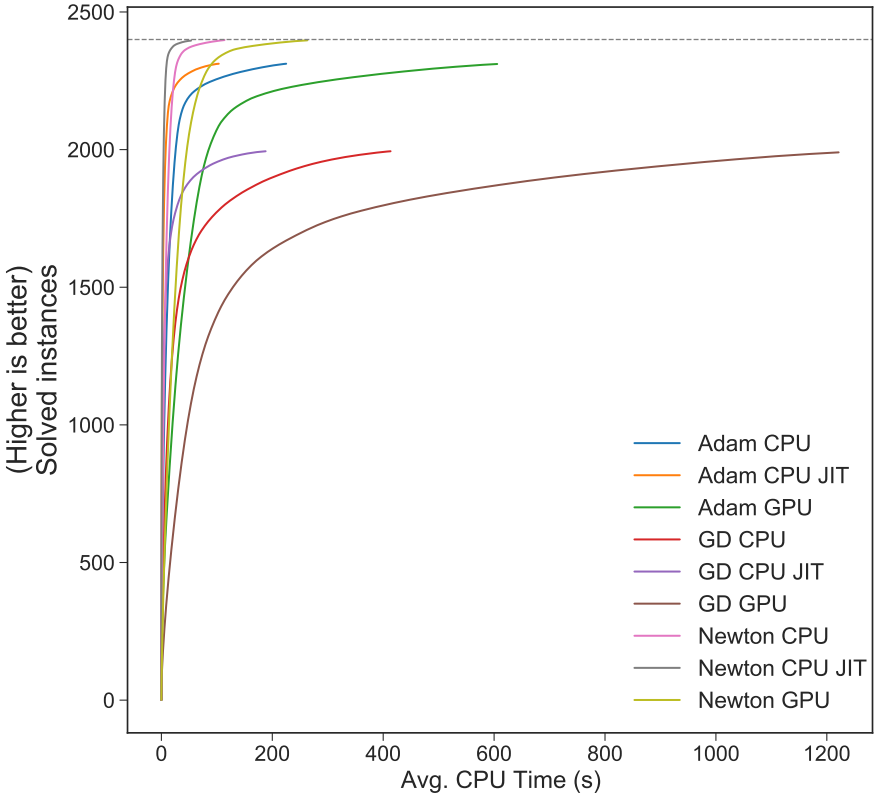


Figure 4.7: Randomly generated programs. CPU: NumPy-based, CPU JIT: Numba-accelerated, GPU: PyTorch, GD: Gradient descent

Discussion

The number of solved instances is plotted against average running time in Figure 4.7. Similarly to the Random Programs experiment, we see the Newton update scheme outperforms Adam and gradient descent in terms of solved instances given max_iter = 100 update budget. Interestingly, in all the update schemes, the GPU implementation was the slowest, and there was more than 5x difference in the running time compared to the fastest (Numba accelerated CPU-JIT). The sizes of program matrices in this task were in the order of hundreds, and the overhead in GPU computation most likely contributed to this result. Increasing the program matrix size, however, brings another issue: when the size of the problem increases, so does the number of restarts (see, for example, Figure 4.2b). While faster iteration is indeed attractive, reducing the number of restarts (and failed iterations) is likely to be more effective in achieving better solving time.

4.5.5 Restart methods

So far, in the experimental section, we only used random sampling from the uniform distribution (Line 2 in Algorithm 4.1). This is another part where we can possibly improve the performance of our method, and in this section we study the effect of using different restart methods. For the implementation of various restart methods, the reader is referred to the previous section.

Experimental details

The programs used in this subsection are a superset of the Random Programs, and are the same as the ones used i n the GPU implementation experiments. We used 3 update methods and aforementioned 5 restart methods, and executed our method 100 times for each program with max_try = 100 and max_iter = 100 resulting in total 3,600,000 data points.

Discussion

Figures 4.8 and 4.9 show the plots of cumulative restarts vs. solved instances at various settings of update and restart method combinations. The general trend with respect to

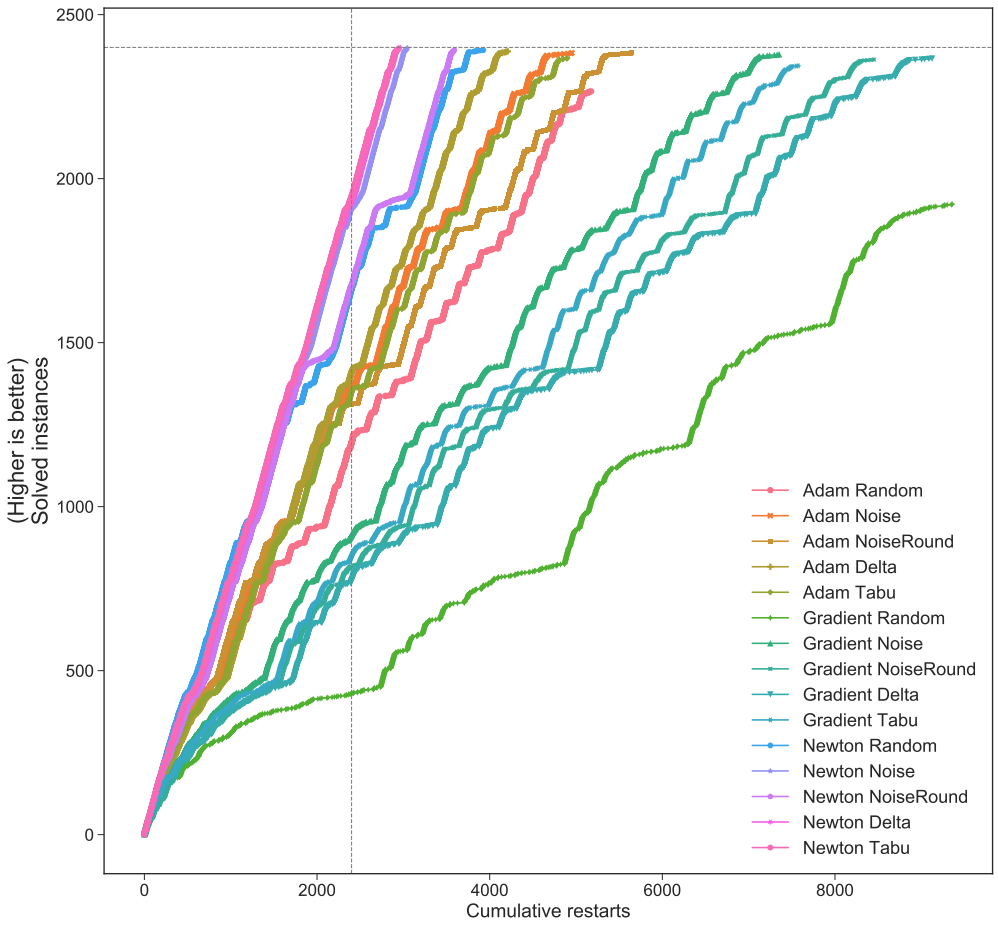
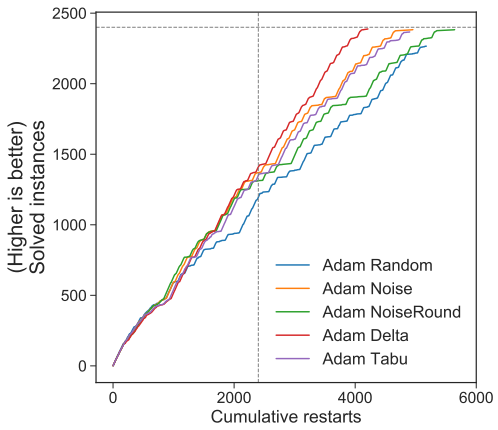
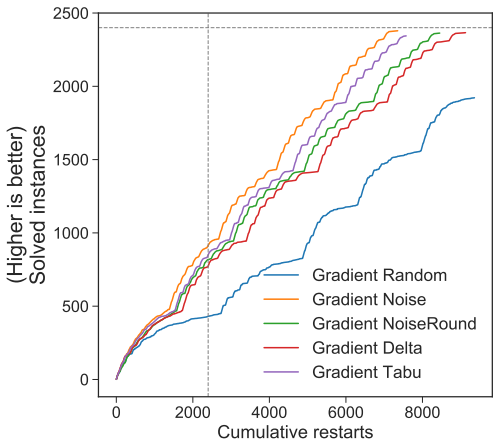


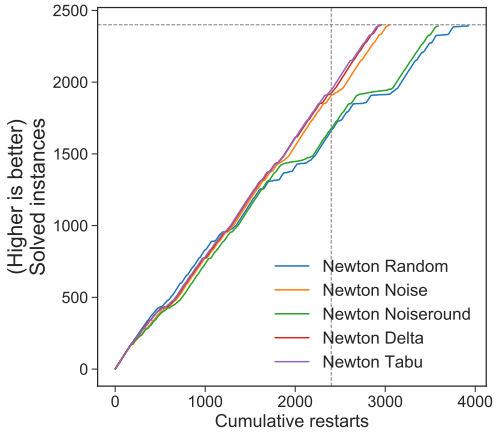
Figure 4.8: Cumulative (mean) number of restarts vs. solved instances, with various update and restart combinations. Failed attempts were excluded.



(a) Cumulative (mean) number of restarts vs. solved instances. Update method = Adam



(b) Cumulative (mean) number of restarts vs. solved instances. Update method = Gradient Descent



(c) Cumulative (mean) number of restarts vs. solved instances. Update method = Newton

Figure 4.9: Cumulative (mean) number of restarts vs. solved instances, grouped by update methods.

Table 4.1: Average time (ms) required to find supported models. Excludes failed attempts. Underlined figures indicate the fastest combination within the same number of atoms.

N.atom	update	Random	Noise	Noise Round	Delta	Tabu
10	Adam	1.74	1.71	**1.47**	1.89	1.95
	GD	8.09	**3.07**	3.70	4.27	3.59
	Newton	**0.63**	0.89	1.00	0.78	0.86
50	Adam	50.33	25.77	31.01	**23.28**	30.99
	GD	134.09	**54.08**	68.97	79.65	59.25
	Newton	16.72	**10.77**	10.97	11.01	10.92
100	Adam	56.31	54.11	65.25	**38.78**	50.28
	GD	123.90	**93.30**	117.93	120.43	93.65
	Newton	37.68	16.25	17.27	**14.71**	15.71
150	Adam	85.82	73.99	82.14	**47.61**	64.04
	GD	174.97	122.22	150.05	151.50	**111.94**
	Newton	48.82	22.82	35.36	21.09	**19.93**
200	Adam	102.63	129.94	137.25	**77.99**	104.34
	GD	214.44	175.73	207.63	217.78	**153.92**
	Newton	70.57	27.95	49.87	**24.49**	26.36

the update methods remained unchanged; the Newton method performed the best overall, followed by Adam and gradient descent. The plots for the restart methods, however, show different characteristics, where the update method can be seen to be reacting differently to the restart method and vice versa. For example, the best performing restart method for Adam was the Delta method, while that for gradient descent was the Noise addition method. It is interesting that, for adaptive gradient update methods (Adam and the Newton update), Delta, Tabu and Noise addition always outperformed Random and Noise rounding, but this was not the case for gradient descent. Moreover, the gap between the three methods (Delta, Tabu and Noise) is small in the Newton update method (Figure 4.9c) while the gap widens in Adam and gradient descent.

Table 4.1 shows the average time (in milliseconds) required to find supported models with various combinations of update and restart methods. For programs with small number of atoms (below 100), the random and noise addition initialization could work well, but it should also be noted that the difference between restart methods is relatively small compared to that in large number of atoms (e.g., N=200). For programs

with more than 100 atoms, the random and noise addition-based methods performed noticeably worse than the Delta and Tabu restart methods. With the Newton update and at N=150 and 200, the random initialization was at least twice as slow as the Delta and Tabu methods.

With the exception of the Newton update at N=10, the random restart did not perform as well as the Delta, Noise addition or Tabu methods. This suggests that when the algorithm fails to find any supported models at the first attempt, there is some benefit to be gained from reusing some parts of the final assignment of the failed attempt. It also suggests we might be able to learn information about the problem on the fly using an online learning algorithm, for example, to improve the performance.

4.5.6 Phase Transition Programs

It has been shown that many NP-hard problems often involve a phase transition, where the difficulty of solving the given problem increases dramatically. For SAT, and in particular 3SAT, it is believed that when the clause-to-variable ratio is around 4.3, the probability of the instance being satisfiable is at 50% [61]. Since many modern ASP solvers also use SAT solvers under the hood, one wonders if similar phase transition can be observed for ASP programs. Thus, Zhao and Lin [62] generated random programs with varying number of rules to atoms ratio, and studied the performance of the then-state-of-the-art ASP solvers. In summary, for programs with rule length 3 (1 head and 2 body literals), they found that when the ratio of number of rules to atoms is around 5, the ASP solvers took longer time to solve the program. Although our algorithm is not meant to be an ASP solver since it finds supported models and not stable models, it would be interesting to see if the phase transition behavior can be observed for our differentiable algorithm.

Experimental details

The random programs were generated according to Zhao and Lin [62]. More specifically, the fixed body length model was used, where L and N denotes the number of rules and atoms, respectively.

To generate a fixed body length program $k - LP(N, L)$, repeat the following steps until L rules are obtained:

1. Generate the head by randomly selecting an atom

2. Generate the body by randomly selecting $(k-1)$ unique atoms, and negate each atom with 50% chance

3. If this rule already exists in the program, discard it and restart from Step 1; otherwise add it to the program.

After L rules were obtained, *clingo* was called to check whether the resulting program has a supported model. If *clingo* returned SAT, then the program was written to a file; otherwise it was discarded and the algorithm restarted from Step 1 with empty rules.

The value of k was fixed at 3 (1 head and 2 body literals), N was selected from $[10, 20, 50, 100, 150]$, L was selected according to the L/N ratio, which was selected at 0.5 interval from 0.5 to 12, and for each setting of L/N, 100 programs were generated. For each N, 2,400 programs were generated, which in total resulted in 12,000 programs.

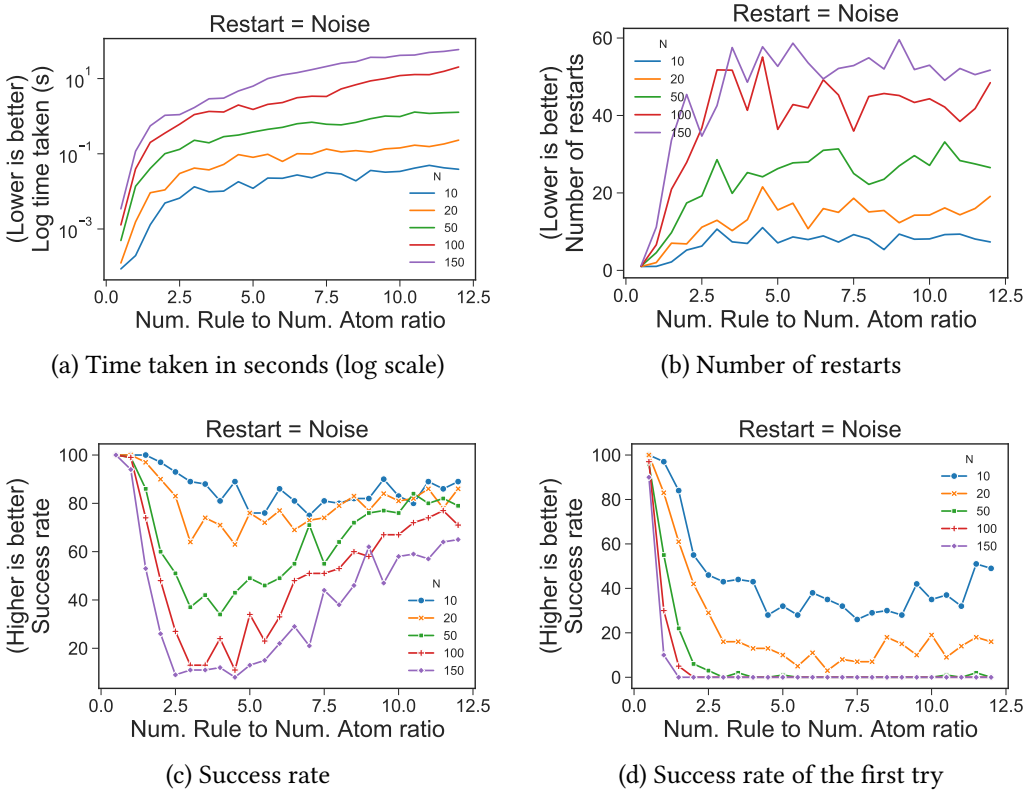
The parameters used in this section are: 1 update method (Newton), 3 restart methods (Noise, Delta, Tabu), max_iter=100 and max_try=100.

Discussion

This phase transition programs contain programs with a wider range of proportions of OR-rules compared to the Random Programs (0.5 to 12 vs 1 to 3). The overall trend in time taken and number of restarts remain unchanged from the previous study on the Random Programs: the number of restarts (and hence the time taken) increases rapidly as soon as OR-rules are added to the program.

Previous study with ASP solvers [62] found a region of hard problems at rules to atoms ratio (L/N) of around 5. Turning to the plots of success rates against (L/N) (figures (c)) in Figures 4.10, 4.11 and 4.12, we can observe that the bottoms of the success rate lines are at around $L/N = 4$. We also note that the "width" of the valley (region where success rate < 100) is wider than that shown in [62].

The first try success rate (figures (d)) graphs show that, again, as soon as we add OR-rules to the program, the success rate of finding the correct models drops rapidly. Also note that, across all restart methods, the programs with larger number of atoms tend to have lower success rate and longer solving time, e.g., the results for N=10 are



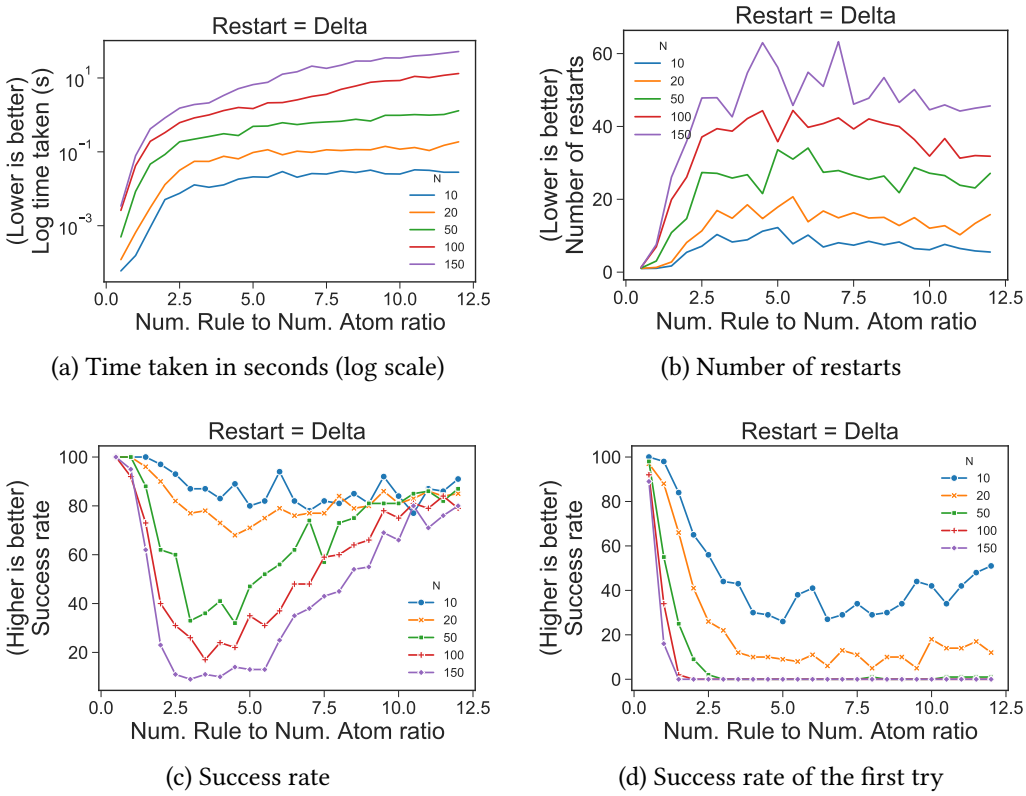
(a) Time taken in seconds (log scale)

(b) Number of restarts

(c) Success rate

(d) Success rate of the first try

Figure 4.10: Phase transition programs, update = Newton, restart = Noise



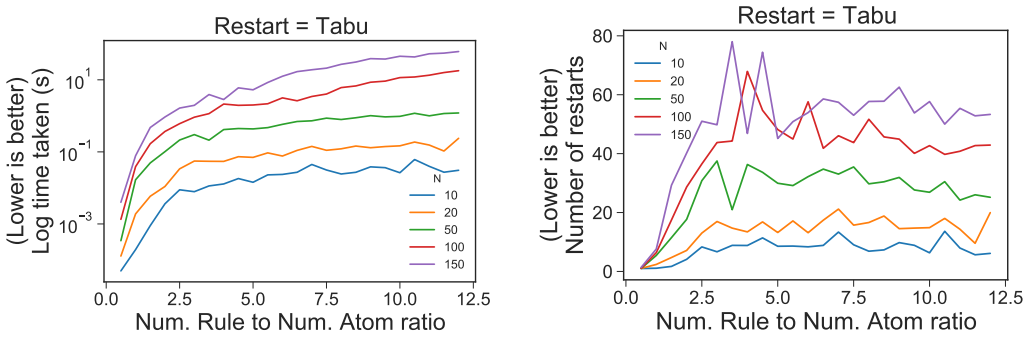
(a) Time taken in seconds (log scale)

(b) Number of restarts

(c) Success rate

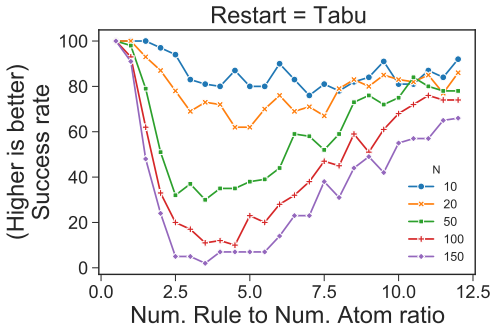
(d) Success rate of the first try

Figure 4.11: Phase transition programs, update = Newton, restart = Delta

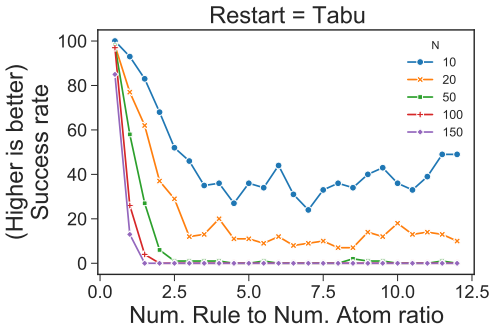


(a) Time taken in seconds (log scale)

(b) Number of restarts



(c) Success rate



(d) Success rate of the first try

Figure 4.12: Phase transition programs, update = Newton, restart = Tabu

consistently better than N=150. Ideally, the algorithm should perform equally well, at least in terms of success rates, for all values of N used in this experiment.

4.6 Conclusion

We presented a new efficient method for the differentiable computation of supported models in continuous vector spaces. The experimental results suggest that our method, when used with adaptive gradient update methods, can find supported models efficiently starting from a random vector. Currently, our method does not support the reduct representation, and we leave differentiable stable model computation for future work. The findings reported in this work are applicable to neuro-symbolic research, especially in the context of enabling logic programming in continuous domains.

5 Conclusion

5.1 Conclusion

In this thesis, we explored various approaches and techniques for the integration of machine learning and knowledge representation. **Chapter 3** presented a method for generating rule sets as global and local explanations for tree-ensemble learning methods using Answer Set Programming. We implemented a decompositional approach where the split structures of the base decision trees were exploited in the construction of rules, which in turn were assessed using pattern mining methods encoded in ASP to extract interesting rules. The main contributions are:

- We present a novel application of Answer Set Programming (ASP) for explaining trained machine learning models. We propose a method to generate explainable rule sets from tree-ensemble models with ASP. More broadly, this work contributes to the growing body of knowledge on integrating symbolic reasoning with machine learning.

- We present how the rule set generation problem can be reformulated as an optimization problem, where we leverage existing knowledge on declarative pattern mining with ASP.
- We show how both global and local explanations can be generated by our approach, while comparative methods tend to focus on either one exclusively.
- To demonstrate the practical applicability of our approach, we provide both qualitative and quantitative results from evaluations with public datasets, where machine learning methods are used in a realistic setting.

Chapter 4 presents a method for computing supported models of normal logic programs in vector spaces using gradient information. First, the program is translated into a definite program and embedded into a matrix representing the program. We introduce a loss function based on the implementation of the immediate consequence operator T_P by matrix-vector multiplication with a suitable thresholding function, and we incorporate regularization terms into the loss function to avoid undesirable results. We report the results of several experiments where our method shows promising performance when used with adaptive gradient update.

The main contributions are:

- Presenting an alternative method for embedding logic programs into matrices, and designing an almost everywhere differentiable thresholding function.
- Introducing a loss function with regularization terms for computing supported models, and integrating various gradient update strategies.
- Demonstrating with a help of systematic performance evaluation on a range of programs, that by selecting appropriate components, it is possible to achieve much higher performance and stability than the existing method.

5.2 Future Work

As for directions for future research, further work could include:

1. ASP for generating explanations of ML models

- Regression and multi-class classification problems
  Current implementation only supports binary classification; however, it can be extended to support regression and multi-class classification problems. Concretely, for supporting regression models, one needs to consider how decision tree ensembles make prediction, then need to account for the role of aggregation functions in generating the explanations. As for the multi-class classification problems, one may extend the current implementation for binary classifiers to support multiple classes in a one-vs-rest setting.

- Counterfactual examples
  For offering more understandable information to the user, one may use counterfactual examples in conjunction with a local explanation for each prediction. In general terms, counterfactual examples show how the decision made by the model can be altered with minimal changes to the input data. This is a challenging task, but one may discretize the input feature space and implement a solution in ASP. The search space for realistic problems is likely to be too large for a naive ASP encoding, however, and one may need to constrain the search space aggressively to solve it in a reasonable amount of time.

- Exploiting multi-shot solving
  The encoding and solving procedure used in this current implementation are rather basic, compared to the full capabilities offered by *clingo*. For any of the future extensions of this framework, such as the ones listed above, one may utilize the advanced solver API offered by *clingo* for efficient solving. For example, one may execute a coarse search around the example using wide intervals to obtain candidate regions before executing a fine-grained search.

2. Differentiable supported model computation

- Effective search and restart strategy
  Current search strategy is essentially a continuous version of random search, and it does not "learn" from failed attempts nor incorporate knowledge about logic programming in the search strategy. A simple online learning

method that adjusts the probability distribution, or any metaheuristic like tabu search may prove to be useful.

- Stable model computation
  With the current method of simulating the fixed point search in continuous space, while one may be able to find stable models when the supported models and stable models coincide, there are cases where this is not the case and thus only supported models are found. The implementation of the immediate consequence (T_P) operator alone is not enough to guarantee that the obtained model is minimal with respect to the program reduct. Thus, one may envision a differentiable method of obtaining stable models, which supports iteration on the program reduct as well as checking for the minimal model configuration.
- Continuous logic programming
  The works presented in this thesis can be applied to domains with continuous interpretations easily. It would be interesting, however, to study the semantics of logic programs in the continuous domain, instead of working exclusively with discrete interpretations.
- Applications in continuous domain
  Past works in the neuro-symbolic literature covered simple image-based tasks such as MNIST addition. Instead, we propose to solve hard combinatoric problems such as Hamiltonian cycle, graph coloring and Sudoku in visual domain. This requires the visual CNN component and the solver part to be trained simultaneously in an end-to-end fashion, and we believe this would lead to much more meaningful implementation of neuro-symbolic architecture.

5.2.1 Towards Integration of ASP-based Explanation Method and Differentiable Logic Programming

As currently implemented, the differences between the projects presented in this thesis makes the integration a non-trivial task. In this subsection, we shall cover some of the challenges when considering the integration of these two projects.

- Domain of the problem
  One of the differences between the tree ensemble learning algorithms and deep neural networks, is the domain of applications. The former tends to be used with tabular datasets, where the input vectors are often represented in a 2D table, whereas the latter are often employed with images, audio tracks, and natural language texts. As of 2022, many of the state-of-the-art results in the latter domains are achieved with some variants of deep neural networks. In our context, the explanations generated by the ASP-based method are human-readable in the sense that the tabular-formatted data often have meaningful column names attached to them. On the other hand, signal-based input data like images have little to no human-comprehensible annotations; it is non-trivial to provide and comprehend a pixel-level definition of a 'cat' for example. Thus, this difference needs to be bridged, or otherwise reconciled in some manner. One may be able to train a tree ensemble model on top of an existing neural network, then generate explanations for it, to be used as prior knowledge for the integrated process. However, this leads to the second challenge, as discussed below.
- Handling of uncertainty and incorrectness
  The explanations generated by our method, and by extension, the tree ensemble learning methods, and decision tree induction algorithms in a machine learning settings, are only approximations to the original model or decision functions. Therefore, the rules are not always *true* in the sense that it is not always 100% accurate. On the other hand, in a typical logic programming setting, the rules can only either be evaluated to *true* or *false*, and the program must not contain contradictory statements; otherwise it becomes inconsistent. In other words, as far as we know, it is not trivial to handle rules that can sometimes be incorrect, in logic programming. Additional work is required to integrate rules from machine learning models into logic programs, and vice versa.

A

Matrix representation of program reduct

The stable model semantics is the basis of Answer Set Programming. Stable models are a subclass of supported models, and have additional requirements. Chapter 4 was broadly based on the supported model semantics, thus in this section, we shall briefly cover the stable model semantics, and introduce our matrix representation of the program reduct. For a more formal introduction to the stable model semantics, refer to Section 2.1.

Consider a normal logic program P:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \neg b_{l+2} \wedge \cdots \wedge \neg b_m (m \geq l \geq 0) \tag{A.1}$$

and the positive and negative occurrences of atoms in the body, $body^+(r) = \{b_1, \ldots, b_l\}$ and $body^-(r) = \{b_{l+1}, \ldots, b_m\}$, respectively.

Informally, the main idea of the stable model semantics is that, given a set of I atoms from the language of P, we simplify P by partially evaluating all rules containing the negated versions of I, then checking whether the simplified program P^I has a Least

Herbrand Model (LHM). The reduct P^I can be obtained procedurally by:

1. delete any rules in P that has a negative literal $\neg b$ in its body where $b \in I$.
2. delete every literal of the form $\neg b$ in the bodies of the remaining rules.

We are now ready to introduce our matrix representation of program reduct.

Definition 13 (Program Reduct Matrix)**.** *Given:*

- P^+*: the positive form of a normal program* P*.*
- $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$*: matrix representation of* P^+*.*
- $\mathbf{Q}_p \in \mathbb{Z}^{N \times N}$ *and* $\mathbf{Q}_n \in \mathbb{Z}^{N \times N}$*: the submatrices of* $\mathbf{Q}$ *that correspond to the positive bodies and negative bodies of the matrix, respectively, such that* $\mathbf{Q} = [\mathbf{Q}_p\, \mathbf{Q}_n]$ *(horizontal concatenation of submatrices).*
- $\mathbf{v} \in \mathbb{Z}^N$*: a subset of an interpretation vector representing* $a_i \in I^P \subseteq B^P_{P^+}$ *if* $v_i = 1$ *for* $\{a_1, \ldots, a_N\}$*.*
- min_1*: thresholding function* $min(x, 1)$*.*
- $I_{hsum=0}(\mathbf{X})$*: an indicator vector where the element is 1 if the sum along the horizontal axis of* $\mathbf{X}$ *is 0.*
- $\mathbf{1}_{dim(\mathbf{x})}$*: a 1-only vector, where the dimension is* $dim(\mathbf{x})$*.*

Then the program reduct P^{+I} *is given by:*

$$P^{+\mathbf{v}} = min_1(\mathbf{Q}_p + diag(I_{hsum=0}(\mathbf{Q}_p))) \odot ((1 - min_1(\mathbf{Q}_n\mathbf{v}))\mathbf{1}^\intercal_{dim(\mathbf{v})}) \tag{A.2}$$

Intuitively, the first term corresponds to the representation of the definite program, where all literals in the form of $\neg b$ are deleted, and the second term corresponds to the first step ("delete any rules ..."). In the first term, $+diag(I_{hsum=0}(\mathbf{Q}_p)))$ is used to preserve facts, when the body becomes empty after removing all negative literals.

Example 3. *Consider the following program:*

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$

We have:

$$\mathbf{Q} = \begin{array}{c} \\ p \\ q \end{array} \begin{array}{c} \begin{array}{cccc} p & q & \overline{p} & \overline{q} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Suppose an interpretation $\mathbf{v}^{\{p\}} = (1\ 0)^\intercal$ *is given. Then, according to the definition, we obtain*

$$\mathbf{P}^{+\mathbf{v}} = \begin{array}{c} \\ p \\ q \end{array} \begin{array}{c} \begin{array}{cc} p & q \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \end{array}$$

which correspond to $p \leftarrow p.$*, and this also happens to be one of the stable models of the program.*

Example 4. *Consider the following program:*

$$p \leftarrow q \wedge \neg r$$
$$q \leftarrow r \wedge \neg p$$
$$r \leftarrow p \wedge \neg q$$

We have:

$$\mathbf{Q} = \begin{array}{r} \\ p \\ q \\ r \end{array} \begin{array}{c} \begin{array}{cccccc} p & q & r & \overline{p} & \overline{q} & \overline{r} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Suppose an interpretation $\mathbf{v}^{\{p,q\}} = (1\ 1\ 0)^\intercal$ *is given. Then, according to the definition, we*

obtain

$$\mathbf{P}^{+\mathrm{v}} = \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{array}{c} \begin{array}{ccc} p & q & r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

which corresponds to $p \leftarrow q$*..*

While one could simply use this representation to compute the fixed-points of the T_P operator, it does not guarantee that the resulting model is *minimal* with respect to the program reduct. Therefore, in order to obtain stable models, another differentiable method for computing minimal models of the program reduct has to be developed, and we leave this for future work.

Bibliography

[1] J. Manyika, J. Silberg, and B. Presten, "What Do We Do About the Biases in AI?," *Harvard Business Review*, Oct. 2019.

[2] UNESCO, "Recommendation on the ethics of artificial intelligence." https://en.unesco.org/artificial-intelligence/ethics. Accessed: 2022-05-03.

[3] D. Gunning, M. Stefik, J. Choi, T. Miller, S. Stumpf, and G.-Z. Yang, "XAI—Explainable artificial intelligence," *Science Robotics*, vol. 4, p. eaay7120, Dec. 2019.

[4] A. Takemura and K. Inoue, "Generating Explainable Rule Sets from Tree-Ensemble Learning Methods by Answer Set Programming," in *Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (Virtual Event), 20-27th September 2021* (A. Formisano, Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens, and N.-F. Zhou, eds.), vol. 345 of *EPTCS*, pp. 127–140, 2021.

[5] A. Takemura and K. Inoue, "Rule Extraction from Decision Tree Ensembles by Answer Set Programming," Sept. 2020. Poster presentation at 17th International Conference on Principles of Knowledge Representation and Reasoning (KR).

[6] A. Takemura and K. Inoue, "Gradient-Based Supported Model Computation in Vector Spaces," in *(to appear) 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2022)*, 2022.

[7] A. Takemura and K. Inoue, "Gradient-Based Supported Model Computation in Vector Spaces," in *Proceedings of the International Conference on Logic Programming*

2021 Workshops Co-Located with the 37th International Conference on Logic Programming (ICLP 2021), Porto, Portugal (Virtual), September 20th-21st, 2021 (J. Arias, F. A. D'Asaro, A. Dyoub, G. Gupta, M. Hecher, E. LeBlanc, R. Peñaloza, E. Salazar, A. Saptawijaya, F. Weitkämper, and J. Zangari, eds.), vol. 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.

[8] K. R. Apt, H. A. Blair, and A. Walker, "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming*, pp. 89–148, Elsevier, 1988.

[9] M. H. Van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM (JACM)*, vol. 23, no. 4, pp. 733–742, 1976.

[10] W. Marek and V. Subrahmanian, "The relationship between stable, supported, default and autoepistemic semantics for general logic programs," *Theor. Comput. Sci.*, vol. 103, no. 2, pp. 365–386, 1992.

[11] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming.," in *ICLP/SLP*, vol. 88, pp. 1070–1080, 1988.

[12] V. Lifschitz, "What is answer set programming?," in *AAAI-08/IAAI-08 Proceedings - 23rd AAAI Conference on Artificial Intelligence and the 20th Innovative Applications of Artificial Intelligence Conference*, pp. 1594–1597, Dec. 2008.

[13] V. Lifschitz, *Answer Set Programming*. Springer International Publishing, 2019.

[14] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + control: Preliminary report," *CoRR*, vol. abs/1405.3694, 2014.

[15] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory and Practice of Logic Programming*, vol. 19, pp. 27–82, Jan. 2019.

[16] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012.

[17] F. Doshi-Velez and B. Kim, "Towards A Rigorous Science of Interpretable Machine Learning," Feb. 2017.

[18] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial Intelligence*, vol. 267, pp. 1–38, Feb. 2019.

[19] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence*, vol. 1, pp. 206–215, May 2019.

[20] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM computing surveys (CSUR)*, vol. 51, no. 5, p. 93, 2019.

[21] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (San Francisco, California, USA), pp. 1135–1144, ACM Press, 2016.

[22] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, pp. 4765–4774, 2017.

[23] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "From local explanations to global understanding with explainable AI for trees," *Nature Machine Intelligence*, vol. 2, pp. 56–67, Jan. 2020.

[24] M. Järvisalo, "Itemset Mining as a Challenge Application for Answer Set Enumeration," in *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 304–310, Springer, 2011.

[25] T. Guyet, Y. Moinard, and R. Quiniou, "Using Answer Set Programming for pattern mining," in *Actes Des Huitièmes Journées de l'Intelligence Artificielle Fondamentale (JIAF'14)*, 2014.

[26] M. Gebser, T. Guyet, R. Quiniou, J. Romero, and T. Schaub, "Knowledge-based sequence mining with ASP," in *Proceedings of the Twenty-Fifth International Joint*

Conference on Artificial Intelligence, IJCAI 2016, pp. 1497–1504, IJCAI/AAAI Press, 2016.

[27] S. Paramonov, D. Stepanova, and P. Miettinen, "Hybrid ASP-based Approach to Pattern Mining," *Theory and Practice of Logic Programming*, vol. 19, no. 4, pp. 505–535, 2019.

[28] L. Breiman and N. Shang, "Born again trees," *University of California, Berkeley, Berkeley, CA, Technical Report*, vol. 1, p. 2, 1996.

[29] S. Hara and K. Hayashi, "Making Tree Ensembles Interpretable: A Bayesian Model Selection Approach," in *International Conference on Artificial Intelligence and Statistics*, pp. 77–85, Mar. 2018.

[30] H. Deng, "Interpreting tree ensembles with intrees," *International Journal of Data Science and Analytics*, vol. 7, no. 4, pp. 277–287, 2019.

[31] B. Liu, W. Hsu, and Y. Ma, "Integrating Classification and Association Rule Mining," in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD '98, (New York, NY), pp. 80–86, AAAI Press, 1998.

[32] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, ICML '95, pp. 115–123, Morgan Kaufmann, 1995.

[33] H. Lakkaraju, S. H. Bach, and J. Leskovec, "Interpretable Decision Sets: A Joint Framework for Description and Prediction," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (San Francisco, California, USA), pp. 1675–1684, ACM Press, 2016.

[34] H. Yang, C. Rudin, and M. I. Seltzer, "Scalable bayesian rule lists," in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70 of *ICML 2017*, pp. 3921–3930, PMLR, 2017.

[35] J. H. Friedman and B. E. Popescu, "Predictive learning via rule ensembles," *The Annals of Applied Statistics*, vol. 2, pp. 916–954, Sept. 2008.

[36] F. Shakerin and G. Gupta, "Induction of Non-Monotonic Logic Programs to Explain Boosted Tree Models Using LIME," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33 of *AAAI '19*, pp. 3052–3059, July 2019.

[37] M. T. Ribeiro, S. Singh, and C. Guestrin, "Anchors: High-precision model-agnostic explanations," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[38] T. Guns, S. Nijssen, and L. De Raedt, "Itemset mining: A constraint programming perspective," *Artificial Intelligence*, vol. 175, pp. 1951–1983, Aug. 2011.

[39] B. Negrevergne, A. Dries, T. Guns, and S. Nijssen, "Dominance programming for itemset mining," in *Proceedings of the 2013 IEEE 13th International Conference on Data Mining*, ICDM '13, pp. 557–566, IEEE, 2013.

[40] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[41] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[42] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems 30*, pp. 3146–3154, Curran Associates, Inc., 2017.

[43] J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen, and B. Baesens, "An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models," *Decision Support Systems*, vol. 51, pp. 141–154, Apr. 2011.

[44] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (San Francisco, California, USA), pp. 785–794, ACM Press, 2016.

[45] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases*, vol. 1215 of *VLDB '94*, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[46] D. Dua and C. Graff, "UCI machine learning repository." https://archive.ics.uci.edu/ml/index.php, 2017.

[47] I. H. Witten, E. Frank, and M. A. Hall, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 2016.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[49] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pp. 2623–2631, 2019.

[50] G. Brewka, J. Delgrande, J. Romero, and T. Schaub, "Asprin: Customizing answer set preferences without a headache," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI '15, pp. 1467–1474, AAAI Press, 2015.

[51] Y. Aspis, K. Broda, A. Russo, and J. Lobo, "Stable and Supported Semantics in Continuous Vector Spaces," in *KR*, pp. 59–68, 2020.

[52] C. Sakama, K. Inoue, and T. Sato, "Linear Algebraic Characterization of Logic Programs," in *KSEM* (G. Li, Y. Ge, Z. Zhang, Z. Jin, and M. Blumenstein, eds.), LNCS, (Cham), pp. 520–533, Springer International Publishing, 2017.

[53] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt, "DeepProbLog: Neural Probabilistic Logic Programming," in *NeurIPS*, pp. 3749–3759, Curran Associates, Inc., 2018.

[54] Z. Yang, A. Ishay, and J. Lee, "NeurASP: Embracing Neural Networks into Answer Set Programming," in *IJCAI-PRICAI 2020*, pp. 1755–1762, 2020.

[55] T. Sato, C. Sakama, and K. Inoue, "From 3-valued Semantics to Supported Model Computation for Logic Programs in Vector Spaces," in *ICAART*, pp. 758–765, 2020.

[56] T. Sato and R. Kojima, "Logical Inference as Cost Minimization in Vector Spaces," in *Artificial Intelligence. IJCAI 2019 International Workshops* (A. El Fallah Seghrouchni and D. Sarne, eds.), LNCS, (Cham), pp. 239–255, Springer International Publishing, 2020.

[57] Y. Dimopoulos and A. Sideris, "Towards Local Search for Answer Sets," in *Logic Programming* (P. J. Stuckey, ed.), LNCS, (Berlin, Heidelberg), pp. 363–377, Springer, 2002.

[58] H. D. Nguyen, C. Sakama, T. Sato, and K. Inoue, "An efficient reasoning method on logic programming using partial evaluation in vector spaces," *J. Log. Comput.*, vol. 31, pp. 1298–1316, 2021.

[59] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural Language Processing (Almost) from Scratch," *J. Mach. Learn. Res.*, vol. 12, no. 76, pp. 2493–2537, 2011.

[60] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[61] D. G. Mitchell, B. Selman, and H. J. Levesque, "Hard and Easy Distributions of SAT Problems," in *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992* (W. R. Swartout, ed.), pp. 459–465, AAAI Press / The MIT Press, 1992.

[62] Y. Zhao and F. Lin, "Answer Set Programming Phase Transition: A Study on Randomly Generated Programs," in *Logic Programming* (C. Palamidessi, ed.), LNCS, pp. 239–253, Springer, 2003.