LOGIC PROGRAMMING-BASED APPROACHES IN EXPLAINABLE ARTIFICIAL INTELLIGENCE AND NATURAL LANGUAGE UNDERSTANDING

by

Farhad Shakerin

APPROVED BY SUPERVISORY COMMITTEE:

Gopal Gupta, Chair

Farokh Bastani

Kevin W. Hamlen

Vibhav Gogate

To Somayeh,

who co-suffered all along

LOGIC PROGRAMMING-BASED APPROACHES IN EXPLAINABLE ARTIFICIAL INTELLIGENCE AND NATURAL LANGUAGE UNDERSTANDING

by

FARHAD SHAKERIN, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2020

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere appreciation and gratitude to my PhD advisor Dr. Gopal Gupta for the continuous support, everlasting encouragement, and optimism that was vital in making this dissertation a reality.

I would like to thank the members of my dissertation defense committee, Dr. Bastani, Dr. Gogate and Dr. Hamlen (in alphabetical order), for providing valued feedback. I greatly thank Dr. Ravi Prakash for career advice, and Dr. Bastani and Dr. Cankaya for writing wonderful recommendation letters.

I sincerely thank Daren and Jennifer Clements for their friendship and support. I genuinely thank my two amazing friends and fellow researchers Kinjal Basu and Sarat Chandra Varanasi for the memories we shared, and for all the useful discussions. I thank my favorite friends Masoud Ghaffarinia and Anahita Mahzari for sharing our sorrow and joy. I also thank my best friend Ali Fakeri Tabrizi for academic and career advice.

Last but not least, I would like to thank my parents Bijan and Ladan and my brother Bahram for their undying love and support.

This dissertation is partially funded by National Science Foundation (NSF) through NSF Grants IIS 1718945 and IIS 1910131. Their support is greatly appreciated.

March 2020

LOGIC PROGRAMMING-BASED APPROACHES IN EXPLAINABLE ARTIFICIAL INTELLIGENCE AND NATURAL LANGUAGE UNDERSTANDING

Farhad Shakerin, PhD
The University of Texas at Dallas, 2020

Supervising Professor: Gopal Gupta, Chair

Dramatic success of machine learning algorithms has led to a torrent of Artificial Intelligence (AI) applications in computer vision and natural language understanding. However, the effectiveness of these systems is limited by the machines' current inability to explain and justify their decisions and actions. The Explainable AI program (Gunning, 2015) aims at creating a suite of machine learning techniques that: a) Produces explainable models without sacrificing predictive performance b) Enables human users to understand the underlying logic and diagnose the mistakes made by the AI system. Inspired by Explainable AI program, this dissertation presents logic programming-based approaches to some of the problems of interest in Explainable AI including learning machine learning hypotheses in the form of default theories, counter-factual reasoning and natural language understanding. In particular, We introduce algorithms that automate learning of default theories. We leverage these algorithms to capture the underlying logic of complex statistical learning models. We also propose a fully explainable logic programming-based framework for visual question answering and introduce a counter-factual reasoner based on Craig Interpolants and Answer Set Programming to come up with recommendations that respect logical, physical, and temporal constraints.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Inductive Logic Programming (ILP) (Muggleton, 1991a) is one Machine Learning technique where the learned model is in the form of logic programming rules (Horn Clauses) that are comprehensible to humans. It allows the background knowledge to be incrementally extended without requiring the entire model to be re-learned. Meanwhile, the comprehensibility of symbolic rules makes it easier for users to understand and verify induced models and even edit them.

Despite all the success in learning a concept by generalization from a set of examples, Inductive Logic Programming still falls short of completely simulating the human common-sense learning. Humans resort to a special kind of reasoning known as default reasoning (Reiter, 1980). Default reasoning gets around the probabilistic calculations by abstracting away the probabilities and by performing default reasoning in the absence of complete knowledge. In Logic Programming, the incomplete knowledge is represented using *negation-as-failure* (NAF). Inductive Logic Programming however, cannot learn default theories as it only handles Horn clauses both in the background knowledge and hypotheses. It turns out that extending ILP with *negation-as-failure* is far from trivial. For one thing, any ILP algorithm capable of handling *negation-as-failure*, must recognise stable model semantics and its implications for the concept learning problems. Stable model semantics, and its realization in answer set programming(ASP), provide an elegant mechanism for handling *negation-as-failure* in logic programming (Gelfond and Lifschitz, 1988; Baral, 2003). Therefore, extending ILP to handle *negation-as-failure*, effectively means to allow background knowledge and/or hypotheses with multiple stable models and this is not trivial in classical ILP.

The primary contributions of this dissertation are the algorithms that automate learning of a concept in the form of default theories. These algorithms can ingest background knowl-

edge in the form of answer set programs with multiple stable models and they can also learn hypotheses in the form of answer set programs with *even loops* through *negation-as-failure* (Baral, 2003). We show important applications for these algorithms including learning ASP programs to solve combinatorial problems.

A major contribution and one of the most important areas that our ILP algorithms can be applied is Explainable AI. We introduce algorithms that leverages our ILP algorithms to capture the global behavior of any complex machine learning model (Shakerin and Gupta, 2019). Explaining the behavior of a machine learning model as default theories has number of advantages over any other rule learning approach: (i) Introducing *negation-as-failure*, significantly decreases the number of induced rules (ii) It significantly improves the performance in terms of the classification evaluation metrics (iii) The induced default theories are easier to understand

Counterfactual explanation is another problem in the field of Explainable AI that has received a lot of attention recently. The EU GDPR regulations (Voigt and Bussche, 2017) requires that machine learning models in charge of making decisions about humans, must provide explanations about the factors that contributed to the decisions they arrive at for any individual. On top of that, they should make recommendations as to how the decision can change to a desired decision. The latter is known as counter-factual explanation problem in explainable AI. For instance, if a machine learning model denies a loan application for an individual, it should explain how that individual needs to minimally change their feature values, so that next time he/she applies, the decision becomes "approved" instead of "denied". All existing approaches solve an instance of optimization problem and by doing so, they leave an important aspect of the problem out, and that is the logical constraints. In this dissertation, we propose the first logic-based approach to the counter-factual explanation problem using answer set programming and Craig-interpolation (Craig, 1957). Our proposed approach, incorporates logical, physical and temporal constraints as part of finding counter-factual explanations.

One of the areas that end-to-end machine learning approaches and in particular deep learning architectures perform well is natural language processing and question answering. As we will show in chapter 9, these systems can even outperform humans in certain domains in visual question answering. In these systems, a set of features are extracted from a convolutional neural network. Then, it is concatenated with a vector representation of the natural language question's words. Finally, the resulted vector is passed through a recurrent neural network and the entire system is trained using time backpropagation algorithm. While the high performance of the system, considering its simplicity and flexibility is quite surprising, it cannot satisfy the minimum expectations of an explainable AI product. On top of that, the system is not robust in the sense that any trivial update to the background knowledge requires the entire system to be trained from the scratch. Moreover, it is almost impossible to explain the answers, spot and diagnose the errors.

In contrast, as we will show in chapter 9, Logic Programming provides an elegant and natural framework for creating a visual question answering system. This framework is fully explainable and provides justification through an SLD resolution proof tree. In this system, the use of machine learning is restricted to the computer vision component that extracts information from a scene and represents them as a logic program. Also, a natural language question is separately translated into a logical representation using a standard off-the-shelf dependency parser. Also, if background knowledge is needed to answer a question, other external sources such as WordNet (Fellbaum, 1998) are consulted. Finally, an answer set programming engine named s(ASP) is used to perform the reasoning and finding the right answer to the natural question. While the results are provably sound, any inconsistency in the answers, whether it is due to the computer vision module, or NLP dependency parsing errors, could be traced all the way back to the source of error. Therefore, our framework is most suitable for visual question answering tasks, as far as the explainability is concerned.

1.2 Structure of the Dissertation

In this section, we provide the layout of the remaining chapters and a summary of each:

Chapter 2 provides background information necessary to understand the remainder of the dissertation. We define the syntax and semantics of ASP programs and introduce the Gelfond-Lifschitz method for finding answer sets. We also discuss Inductive Logic Programming (ILP) problem, default reasoning, and finally, we survey some of the explainable AI techniques that are leveraged by our ILP algorithms.

Chapter 3 introduces a heuristic-based ILP algorithm FOLD (First-Order Learner of Default-theories) that learns a concept from examples and background knowledge in the form of default theories. This algorithm learns stratified answer set programs (i.e., answer set programs with one stable model)

Chapter 4 extends the FOLD algorithm to learn Answer Set Programs with multiple stable models. We showcase applications in learning programs that would solve combinatorial puzzles.

Chapter 5 introduces our Explainable AI (XAI) contribution, which is an extension of FOLD algorithm to learn the underlying logic of any complex machine learning model in terms of default theories.

Chapter 6 proposes an Explainable AI algorithm to capture the underlying logic of Support Vector Machines with different Kernels. This algorithm mostly focuses on Support Vectors and the similarities of any given data point to Support Vector points.

Chapter 7 enhances the search for the "best" clause in LIME-FOLD algorithm and also replaces LIME with SHAP which is a sound Explainable AI tool with foundations in game theory. Also, instead of performing *hill-climbing* search using *information-gain*, SHAP-FOLD algorithm incorporates a data mining technique known as High-Utility Itemset Mining (HUIM) to find frequent patterns with high utilities.

Chapter 8 proposes our logic based solution to another requirement stipulated in GDPR and Explainbale AI project known as counter-factual explanation. In this chapter we show how using Answer Set Programming and Craig Interpolants, we can create counter-factual explanations that respect logical, physical, and temporal constraints that are also expressed as answer set programs.

Chapter 9 illustrates our fully explainable framework AQuA, based on answer set programming to tackle the visual question answering task. Our AQuA framework, achieves competitive performance results with end-to-end neural network based solutions, but unlike those systems, AQuA is fully explainable and it can provide justification for all the answers it finds, it requires no training beyond the computer vision component and one can diagnose wrong answers and find the source of error and fix it.

Finally, in Chapter 10 we present some avenues for the future works and the conclusions.

CHAPTER 2

BACKGROUND

2.1 Overview

As most of the algorithms presented in this dissertation are based on Answer Set Programming and since they address problems in Inductive Logic Programming, understanding of both concepts is imperative to comprehend the remainder of this dissertation. In this chapter we provide a brief overview of each topic.

2.2 Answer Set Programming

Answer Set Programming (Gelfond and Lifschitz, 1988; Baral, 2003) is a declarative logic programming based paradigm that deals with *negation-as-failure*. The main distinction between standard Logic Programming and Answer Set Programs is the interpretation of the loops through *negation-as-failure* as follows:

```
p :- not q.
q :- not p.
```

This program does not have any semantics based on SLD resolution because it falls in an infinite loop. However, in stable model semantics which is the foundation of Answer Set Programming, this program indeed has a meaning. Next, we will describe the syntax of ASP programs:

Definition 2.1. *An* ***atom*** *is a predicate of the form* $p(x_1, ..., x_n), n \geq 0$*, where each* x_i *is a constant integer or string. When* $n = 0$ *parentheses are omitted. A* ***negated*** *atom is an atom that is preceded by* `not`*.*

Definition 2.2. *A* ***literal*** *is an atom or its negation.*

Definition 2.3. *A **clause** is of the following form:*

1. $l_0 :- l_1, ..., l_n.$

2. $:- l_0, ..., l_n.$

Each clause has two parts: head and body. The *head* and the *body* part could be empty. If *body* is empty, the rule is called a fact. Clauses of the form (2) are headless and are called integrity constraints. They are treated as if `false` is in the *head*. The head succeeds only if every literal in the body succeeds. For instance, in the clause `p :- not q, r.`, the predicate `p` succeeds if `not q` and `r` both succeed.

Definition 2.4. *A **normal** logic program is a finite set of clauses defined in Definition 2.3.*

With the exception of s(ASP) system (Marple et al., 2017), all current ASP engines require an ASP program to be *grounded*. Grounding is the process of eliminating variables and replacing them with all possible combinations of constants. For example given the following clause:

```
r :- p(X), q(Y).
```

where `X` and `Y` can each be bound to 1 and 2, grounding will result in the following set of propositional clauses:

```
r :- p(1), q(1).
r :- p(1), q(2).
r :- p(2), q(1).
r :- p(2), q(2).
```

There are various semantics for *negation-as-failure*. In this dissertation, we follow the *Stable Model Semantics* by (Gelfond and Lifschitz, 1988). In this semantics `not p` holds if we fail

to establish a proof for `p`. Therefore, the semantics of an ASP program is defined in terms of the transformation known as *Gelfond-Lifschitz Transformation* or GL method for computing the stable models of an ASP program:

Definition 2.5. ***Gelfond-Lifschitz Transformation*** *For a grounded ASP program P and a potential stable model A, a residual program R is created as follows: for each literal $L \in A$:*

1. *Remove any clause in P with* `not L` *in the body.*
2. *Remove any negative literals from the remaining clauses' bodies.*

Let F be the least fixed-point semantics of R. If $F = A$, then the potential stable model A is indeed a stable model of P.

Finding the set of stable models of an ASP program P is an NP complete problem and therefore, the GL method based on guess and check is just about the best method to find the set of all stable models for P.

Example 2.1. *We find all stable models for the following program:*

```
(1)    p :- not q.
(2)    q :- not p.
```

Set of all candidate stable models includes {{p},{q},{p,q},{}}.
For $A = \{p\}$, clause (2) is removed,because, according to GL method rule #1, the body contains `not p`. Then, following the GL method's rule #2, `not q` is removed from the body of clause (1). The resulting residual program R's fixed-point is the set $F = \{p\}$. Since, $F = A$, therefore, the potential stable model A is indeed a stable model for the program. Similarly, we can show that $\{q\}$ is another stable model for the program. For $A = \{p, q\}$, both clauses will be removed and therefore, the fixed-point of residual program R will become the empty set. Thus, $\{p, q\}$ is not a stable model for the original program.

Headless clauses or integrity constraints in ASP are clauses of the following form:

```
:- q1, ..., qn.
```

which equivalently could be re-written as:

```
p :- q, not p.
q :- q1,...,qn.
```

Under the stable model semantics, `q` is effectively forced to be false in any stable model, because, otherwise, if `p` belongs to a stable model, the entire clause will be removed, and if `p` does not belong to the dataset, according to rule #2 of GL method, the only way to prevent a contradiction by having both `p` and `not p` in the same stable model is to force `q` to fail.

2.3 Default Theories

Default Logic (Reiter, 1980) is a *non-monotonic* logic to formalize reasoning with default assumptions. Normal logic programs provide a simple and practical formalism for expressing default rules. A default rule of the form $\frac{\alpha_1 \wedge ... \wedge \alpha_m : \neg\beta_{m+1}, ..., \neg\beta_n}{\gamma}$ can be formalized as the following normal logic program:

$$\gamma \leftarrow \alpha_1, ..., \alpha_m, not\ \beta_{m+1}, ..., not\ \beta_n$$

where γ, αs and βs are positive predicates.

2.4 Inductive Logic Programming

The problem that we tackle in this dissertation is an inductive non-monotonic logic programming problem which can be formalized as follows:

Given

- a background theory $\mathcal{B}$, in the form of an extended logic program, i.e, clauses of the form $h \leftarrow l_1, ..., l_m, not\ l_{m+1}, ..., not\ l_n$. where $l_1, ...l_n$ are positive literals and *not* denotes *negation-as-failure (NAF)* with stable model semantics;

- two disjoint sets of grounded goal predicates $\mathcal{E}^+$,$\mathcal{E}^-$, known as positive and negative examples respectively;
- a hypothesis language of predicates $\mathcal{L}$ including function and atom free predicates. It also contains a set of arithmetic constraints of the form $\{A \leq h, A \geq h\}$ where A is a variable and h is a real number;
- a $covers(\mathcal{H}, \mathcal{E}, \mathcal{B})$ function, which returns the subset of $\mathcal{E}$ which is extensionally implied by the current hypothesis $\mathcal{H}$ given the background knowledge B;
- a $score(\mathcal{E}^+, \mathcal{E}^-, \mathcal{H}, \mathcal{B})$ function, which specifies the quality of the hypothesis $\mathcal{H}$ with respect to $\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}$;

Find

- a theory $\mathcal{T}$ for which $covers(\mathcal{T}, \mathcal{E}^+, \mathcal{B})$ is just $\mathcal{E}^+$ and $covers(\mathcal{T}, \mathcal{E}^-, \mathcal{B})$ is $\emptyset$.

CHAPTER 3

INDUCTIVE LEARNING OF DEFAULT THEORIES

3.1 Overview

Predictive models produced by classical machine learning methods are not comprehensible for humans because they are algebraic solutions to optimization problems such as risk minimization or data likelihood maximization. These methods do not produce any intuitive description of the learned model. This makes it hard for users to understand and verify the underlying rules that govern the model. As a result, these methods do not produce any justification when they are applied to a new data sample. Also, extending the prior knowledge[1] in these methods requires the entire model to be re-learned. Additionally, no distinction is made between *exceptions* and noisy data. *Inductive Logic Programming* (Muggleton, 1991b), however, is one technique where the learned model is in the form of logic programming rules (Horn clauses) that are more comprehensible and that allows the background knowledge to be incrementally extended without requiring the entire model to be relearned. This comprehensibility of symbolic rules makes it easier for users to understand and verify the resulting model and even edit the learned knowledge.

Given the background knowledge and a set of positive and negative examples, ILP learns theories in the form of Horn logic programs. However, Horn clauses are not sufficiently expressive for representation and reasoning when the background knowledge is incomplete.

Additionally, ILP is not able to handle exception to general rules: it learns rules under the assumption that there are no exceptions to them. This results in exceptions and noise being treated in the same manner. Often, the exceptions to the rules themselves follow a pattern, and these exceptions can be learned as well. The resulting theory that is learned is a default

[1]In the rest of the chapter we will use the term background knowledge to refer to prior knowledge (Muggleton, 1991b).

theory, and in most cases this theory describes the underlying model more accurately. It should be noted that default theories closely model common sense reasoning as well (Baral, 2003). Thus, a default theory, if it can be learned, will be more intuitive and comprehensible for humans. Default reasoning also allows us to reason in absence of information. A system that can learn default theories can therefore learn rules that can draw conclusions based on lack of evidence, just like humans. Other reasons that underscore the importance of inductive learning of default theories can be found in Sakama (Sakama, 2005) who also surveys other attempts in this direction.

As an example, suppose we want to learn the concept of flying ability of birds. We would like to learn the default rule that birds normally fly, as well as rules that capture exceptions, namely, penguins and ostriches are birds that do not fly. Current ILP systems will be thrown off by the exceptions and will not discover any general rule: they will just either enumerate all the birds that fly or cover the positive examples without caring much about the falsely covered negative examples. Other algorithms, such as FOIL, will induce rules that are non-constructive and thus not helpful or intuitive.

In this chapter, we present two algorithms for learning default theories (i.e., non monotonic logic programs), called FOLD (First Order Learner of Default) and FOLD-R, to handle categorical and numeric features respectively. Unlike traditional ILP systems that learn standard logic programs (i.e., no negation is allowed), our algorithms learn *non monotonic logic programs* (that allow *negation-as-failure*). Our algorithms are an extension of the FOIL algorithm (Quinlan, 1990a) and support both categorical and numeric features. Whenever needed, our algorithms introduce new predicates. The language bias (Mitchell, 1980) also contains arithmetic constraints of the form $\{A \leq h, A \geq h\}$. The algorithms have been implemented and tried on variety of datasets from the UCI repository. They have shown excellent results that are presented here as well.

The default theories that we learn using our algorithm, as well as the background knowledge used, is assumed to follow the stable model semantics. Stable model semantics, and its

realization in answer set programming(ASP), provides an elegant mechanism for handling negation in logic programming (Gelfond and Lifschitz, 1988). We assume that the reader is familiar with ASP and stable model semantics (Baral, 2003).

This chapter makes the following contributions: We propose a novel concrete algorithm to learn default theories automatically in the absence of complete information. The proposed algorithm, unlike the existing ones, is able to handle the numeric features without discretizing them first, and is also capable of handling non-monotonic background knowledge. We provide both qualitative and quantitative results from standard UCI datasets to support the claim that our algorithm discovers more accurate as well as more intuitive rules compared to the conventional ILP systems.

3.2 Background

Our algorithm to learn default theories is an extension of the FOIL algorithm (Quinlan, 1990a). FOIL is a top-down ILP system which follows a *sequential covering* approach to induce a hypothesis. The FOIL algorithm is summarized in Algorithm 1. This algorithm repeatedly searches for clauses that score best with respect to a subset of positive and negative examples, a current hypothesis and a heuristic called *information gain* (IG).

The inner loop searches for a clause with the highest information gain using a general-to-specific hill-climbing search. To specialize a given clause c, a refinement operator ρ under θ-subsumption (Plotkin, 1971) is employed. The most general clause is $p(X_1, ..., X_n) \leftarrow true$. where the predicate p/n is the predicate being learned and each X_i is a variable. The refinement operator specializes the current clause $h \leftarrow b_1, ...b_n$. This is realized by adding a new literal l to the clause yielding $h \leftarrow b_1, ...b_n, l$. The heuristic based search uses information gain. In FOIL, information gain for a given clause is calculated as follows (Mitchell, 1997):

$$IG(L, R) = t\left(log_2\frac{p_1}{p_1 + n_1} - log_2\frac{p_0}{p_0 + n_0}\right) \tag{3.1}$$

Algorithm 1 Summarizing the FOIL algorithm

Input: $goal, \mathcal{B}, \mathcal{E}^+, \mathcal{E}^-$
Output: Initialize $\mathcal{H} \leftarrow \emptyset$

```
1: while (stopping criterion) do
2:     c ← (goal :- true.)
3:     while (stopping criterion) do
4:         for all c' ∈ ρ(c) do
5:             compute score(E+, E−, H ∪ {c'}, B)
6:         end for
7:         let ĉ be the c' ∈ ρ(c) with the best score
8:     end while
9:     add ĉ to H
10:    E+ ← E+ \ covers(ĉ, E+)
11: end while
```

where L is the candidate literal to add to rule R, p_0 is the number of positive bindings of R, n_0 is the number of negative bindings of R, p_1 is the number of positive bindings of $R + L$, n_1 is the number of negative bindings of $R+L$, t is the number of positive bindings of R also covered by $R+L$. FOIL handles negated literals in a naive way by adding the literal *not* L to the set of specialization candidate literals for any existing candidate L. This approach leads to learning predicates that do not capture the concept accurately as shown in the following example.

Example 3.1. $\mathcal{B}, \mathcal{E}^+$ *are background knowledge and positive examples respectively with CWA and the concept to be learned is fly.*

$\mathcal{B}$: $bird(X) \leftarrow penguin(X).$

$bird(tweety).$ $bird(et).$

$cat(kitty).$ $penguin(polly).$

$\mathcal{E}^+$: $fly(tweety).$ $fly(et).$

The FOIL algorithm would learn the following rule:

```
fly(X) :- not cat(X), not penguin(X).
```

which does not yield a constructive definition even though it covers all the positives (tweety and et are not penguins and cats resp.) and no negatives (neither cats nor penguins do not fly). In fact, the correct theory in this example is as follows: "*Only birds fly but, among them there are exceptional ones who do not fly*". It translates to the following Prolog rule:

```
fly(X) :- bird(X), not penguin(X).
```

which FOIL fails to discover.

3.3 FOLD Algorithm

The idea of our FOLD algorithm is to learn a concept as a default and possibly multiple exceptions. In that sense, FOLD tries first to learn the default by specializing a general rule of the form $goal(V_1, ..., V_n) \leftarrow true.$ with positive literals. As in FOIL, each specialization must rule out some already covered negative examples without decreasing the number of positive examples covered significantly. Unlike FOIL, no negative literal is used at this stage. Once the IG becomes zero, this process stops. At this point, if some negative examples are still covered, they must be either noisy data samples or exceptions to the so far learned rule. As (Srinivasan et al., 1996) discuss, there is no pattern distinguishable in noise, whereas, in exceptions, there may exist a pattern that can be described using the same language bias. This can be viewed as a subproblem to (recursively) find the rules governing a bunch of negative examples. To achieve that aim, FOLD swaps the current positive and negative examples and recursively calls the FOLD algorithm to learn the exception rule(s). Each time a rule is discovered for exceptions, a new predicate $ab(V_1, ..., V_n)$ is introduced. To avoid name collision, FOLD appends a unique number at the end of the string ab to guarantee the uniqueness of the invented predicates.

In case of noisy data or in the presence of uncertainty due to the lack of information, it turns out that there is no pattern to learn. In such cases, FOLD enumerates the positive

examples for two purposes: first, this is essential for the training algorithm to converge, second, it helps to detect noisy data samples.

Algorithm 2 shows a high level implementation of the FOLD algorithm. In lines 1-8, function FOLD, serves as the FOIL outer loop. In line 3, FOLD starts with the most general clause (e.g $fly(X) \leftarrow true.$). In line 4, this clause is refined by calling the function $SPECIALIZE$. In lines 5-6, set of positive examples and set of discovered clauses are updated to reflect the newly discovered clause. In lines 9-29, the function $SPECIALIZE$ is shown. It serves as the FOIL inner loop. In line 12, by calling the function ADD_BEST_LITERAL the "best" positive literal is chosen and the best IG as well as the corresponding clause is returned. In lines 13-24, depending on the IG value, either the positive literal is accepted or the EXCEPTION function is called. If, at the very first iteration, IG becomes zero, then a clause that just enumerates the positive examples is produced. A flag called *just_started* handles this checking. In lines 26-27, the sets of positive and negative examples are updated to reflect the changes of the current clause. In line 19, the EXCEPTION function is called while swapping the $\mathcal{E}^+, \mathcal{E}^-$.

In line 31, we find the "best" positive literal that covers more positive examples and fewer negative examples. Again, note the current positive examples are really the negative examples and in EXCEPTION function, we try to find the rule(s) governing the exception. In line 33, FOLD is recursively called to extract this rule(s). In line 34, a new *ab* predicate is introduced and in lines 35-36 it is associated with the body of the rule(s) found by the recurring FOLD function call in line 33. Finally, in line 38, default and exception are attached together to form a single clause.

The FOLD algorithm, once applied to Example 3.1 yields the following clauses:

```
fly(X) :- bird(X), not ab0(X).
ab0(X) :- penguin(X).
```

Algorithm 2 FOLD Algorithm

Input: $goal, \mathcal{B}, \mathcal{E}^+, \mathcal{E}^-$

function FOLD($\mathcal{E}^+, \mathcal{E}^-$)
  while ($size(\mathcal{E}^+) > 0$) **do**
    $c \leftarrow (goal$:- $true.)$
    $\hat{c} \leftarrow$ SPECIALIZE($c, \mathcal{E}^+, \mathcal{E}^-$)
    $\mathcal{E}^+ \leftarrow \mathcal{E}^+ \setminus covers(\hat{c}, \mathcal{E}^+)$
    $D \leftarrow D \cup \{\hat{c}\}$
  end while
end function
function SPECIALIZE($c, \mathcal{E}^+, \mathcal{E}^-$)
  $just\_started \leftarrow true$
  while ($size(\mathcal{E}^-) > 0$) **do**
    $(c_{def}, \hat{IG}) \leftarrow$ ADD_BEST_LITERAL($c, \mathcal{E}^+, \mathcal{E}^-$)
    if $\hat{IG} > 0$ **then**
      $\hat{c} \leftarrow c_def$
    else
      if $just\_started$ **then**
        $\hat{c} \leftarrow enumerate(c, \mathcal{E}^+)$
      else
        $\hat{c} \leftarrow$ EXCEPTION($c, \mathcal{E}^-, \mathcal{E}^+$)
        if $\hat{c} = null$ **then**
          $\hat{c} \leftarrow enumerate(c, \mathcal{E}^+)$
        end if
      end if
    end if
    $just\_started \leftarrow false$
    $\mathcal{E}^- \leftarrow \mathcal{E}^- \setminus covers(\hat{c}, \mathcal{E}^-)$
  end while
end function
function EXCEPTION($c_{def}, \mathcal{E}^+, \mathcal{E}^-$)
  $\hat{IG} \leftarrow$ ADD_BEST_LITERAL($c, \mathcal{E}^+, \mathcal{E}^-$)
  if $\hat{IG} > 0$ **then**
    $c\_set \leftarrow$ FOLD($\mathcal{E}^+, \mathcal{E}^-$)
    $c\_ab \leftarrow generate\_next\_ab\_predicate()$
    for each $c \in c\_set$ **do**
      $AB \leftarrow AB \cup \{c\_ab$:- $bodyof(c)\}$
    end for
    $\hat{c} \leftarrow (headof(c_{def})$:- $bodyof(c), \mathbf{not}(c\_ab))$
  else
    $\hat{c} \leftarrow null$
  end if
end function

Now, we illustrate how FOLD discovers the above set of clauses given $\mathcal{E}^+ = \{tweety, et\}$ and $\mathcal{E}^- = \{polly, kitty\}$ and the goal $fly(X)$. By calling FOLD, in line 2 "while", the clause $fly(X) \leftarrow true.$ is specialized. In $SPECIALIZE$ function, in line 12, the literal $bird(X)$ is picked to add to the current clause, to get the clause $\hat{c} = fly(X) \leftarrow bird(X)$ which happened to have the greatest IG among $\{bird, penguin, cat\}$. Then, in line 26-27 the following updates are performed: $\mathcal{E}^+ = \{\}$,$\mathcal{E}^- = \{polly\}$. A negative example $polly$, a penguin is still covered. In the next iteration, $SPECIALIZE$ fails to introduce a positive literal to rule it out since the best IG in this case is zero. Therefore, the EXCEPTION function is called by swapping the $\mathcal{E}^+$, $\mathcal{E}^-$. Now, FOLD is recursively called to learn a rule for $\mathcal{E}^+ = \{polly\}$, $\mathcal{E}^- = \{\}$. The recursive call (line 33), returns $fly(X) \leftarrow penguin(X)$ as the exception. In line 34 a new predicate $ab0$ is introduced and in line 35-37 the clause $ab0(X) \leftarrow penguin(X)$ is created and added to the set of invented abnormalities namely, AB. In line 38, the negated exception (i.e not $ab0(X)$) and the default rule's body (i.e $bird(X)$) are compiled together to form the clause $fly(X) \leftarrow bird(X), not\ ab0(X)$.

Note, in two different cases *enumerate* is called. First, at very first iteration of specialization if IG is zero for all the positive literals. Second, when the *Exception* routine fails to find a rule governing the negative examples. Whichever is the case, corresponding samples are considered as noise. The following example shows the learned logic program in presence of noise.

Example 3.2. *Similar to Example 2.1, plus we have an extra positive example fly(jet) without any further information:*

$\mathcal{B}:$ $bird(X) \leftarrow penguin(X).$

$bird(tweety).$ $bird(et).$

$cat(kitty).$ $penguin(polly).$

$\mathcal{E}^+:$ $fly(tweety).$ $fly(jet).$ $fly(et).$

FOLD algorithm on the Example 3.2 yields the following clauses:

```
fly(X) :- bird(X), not ab0(X).
fly(X) :- member(X,[jet]).
ab0(X) :- penguin(X).
```

FOLD recognizes *jet* as a noisy data. *member*/2 is a built-in predicate in SWI-Prolog to test the membership of an atom in a list.

Sometimes, there are nested levels of exceptions. The following example shows how FOLD manages to learn the correct theory in presence of nested exceptions.

Example 3.3. *Birds and planes normally fly, except penguins and damaged planes that can't. There are super penguins who can, exceptionally, fly.*

$\mathcal{B}$: $bird(X) \leftarrow penguin(X).$

$penguin(X) \leftarrow superpenguin(X).$

$bird(a).\quad bird(b).\quad penguin(c).\quad penguin(d).$

$superpenguin(e).\quad superpenguin(f).\quad cat(c1).$

$plane(g).\quad plane(h).\quad plane(k).\quad plane(m).$

$damaged(k).\quad damaged(m).$

$\mathcal{E}^+$: $fly(a).\quad fly(b).\quad fly(e).$

$fly(f).\quad fly(g).\quad fly(h).$

FOLD algorithm learns the following theory:

```
fly(X) :- plane(X), not ab0(X).
fly(X) :- bird(X), not ab1(X).
fly(X) :- superpenguin(X).
ab0(X) :- damaged(X).
ab1(X) :- penguin(X).
```

Theorem 3.1. *The FOLD algorithm terminates on any finite set of examples.*

Proof. It suffices to show that the size of $\mathcal{E}^+$ on every iteration of FOLD function decreases (at line 5) and since $\mathcal{E}^+$ is a finite set, it will eventually becomes empty and the while loop terminates. Equivalently, we can show that every time the SPECIALIZE function is called, it terminates and a clause $\hat{c}$ that covers a non-empty subset of $\mathcal{E}^+$ is returned. Inside SPECIALIZE function, if $\mathcal{E}^-$ is empty, then the function returns its input clause and the theorem trivially holds. Otherwise, two cases might happen: First, it produces a clause which enumerates $\mathcal{E}^+$ and covers no negative example, returns immediately and again the theorem trivially holds. Second, it calls the EXCEPTION function which may lead to a chain of recursive calls on FOLD function. In this case it suffices to show that on a chain of recursive calls on FOLD, the size of function argument i.e. $\mathcal{E}^+$ decreases each time. That's indeed the case because every time a literal is added to the current clause in line 12, it decreases the size of covered negative examples from $\mathcal{E}^-$, which in turn becomes the new $\mathcal{E}^+$ as the EXCEPTION function and subsequently the FOLD function is called. Therefore, on consecutive calls to FOLD function, the size of input argument $\mathcal{E}^+$ is decreased hence it eventually terminates. □

3.4 Numeric Extension of FOLD

ILP systems have limited application to data sets containing a mix of categorical and numerical features. A common way to deal with numerical features is to discretize the data to qualitative values. This approach leads to accuracy loss and requires domain expertise. Instead, we adapt the approach taken in the well-known C4.5 algorithm (Quinlan, 1993). This algorithm is ranked no. 1 in the survey paper "*Top 10 algorithms in datamining*", (Wu et al., 2007). For a numeric feature A, constraints such as $\{A \leq h, A > h\}$ have to be considered where the threshold h is found by sorting the values of A and choosing the split between successive values that maximizes the information gain. In our FOLD-R algorithm that we propose and describe next, we perform the same method for a set of operators $\{<, \leq\}$ and

pick the operator and threshold which maximizes the information gain. Also, we need to extend the ILP language bias to support the arithmetic constraints.

Unlike the categorical features for which we use *propositionalization* (Kramer et al., 2000) ,for numeric features we define a predicate that contains an extra variable which always pairs with a constraint. For example to extend the language bias for a numeric quantity "age" we could define predicates of the form $age(a, b)$ in the background knowledge, and the candidate to specialize a clause might be as follows: $age(X, N), N \leq 5$. However, the predicate $age/2$ never appears without the corresponding constraint.

Algorithm 3 illustrates the high level changes made to FOLD, in order to obtain the FOLD-R algorithm. The function *test_categorical*, as before, chooses the best categorical literal to specialize the current clause. The function *test_numeric* chooses the best numeric literal as well as the best arithmetic constraint and threshold with the highest IG. In line 5, if neither one leads to a positive IG, exception is tried. If exception also fails, then *enumerate* is called. Otherwise, IGs are compared and whichever is greater, the corresponding clause is chosen as the specialized clause of the current iteration.

Example 3.4. *Table* **??** *adapted from (Quinlan, 1993) is a dataset with numeric features "temperature" and "humidity". "Outlook" and "Windy" are categorical features. Our FOLD-R algorithm, for the goal play*(X)*, and positive examples shown as records with label "Play", and negative examples shown as records with label "Don't Play" outputs the following clauses:*

```
play(X) :- overcast(X).
play(X) :- temperature(X, A), A <= 75, not ab0(X).
ab0(X) :- windy(X), rainy(X).
ab0(X) :- humidity(X, A), A >= 95, sunny(X).
```

FOLD-R results suggest an abnormal day to play is either a rainy and windy day or a sunny day with above 95% humidity.

Algorithm 3 FOLD-R Algorithm, Specialize function. The rest of the functions remain unchanged

```
1: function SPECIALIZE(c, E+, E-)
2:     while (size(E-) > 0) do
3:         (ĉ1, IĜ1) ← test_categorical(c, E+, E-)
4:         (ĉ2, IĜ2) ← test_numeric(c, E+, E-)
5:         if IĜ1 = 0 & IĜ2 = 0 then
6:             ĉ ← EXCEPTION(c, E-, E+)
7:             if ĉ = null then
8:                 ĉ ← enumerate(c, E+)
9:             end if
10:        else
11:            if IĜ1 ≥ IĜ2 then
12:                ĉ ← ĉ1
13:            else
14:                ĉ ← ĉ2
15:            end if
16:        end if
17:        E- ← E- \ covers(ĉ, E-)
18:    end while
19: end function
```

Table 3.1: Play Tennis data-set, Numeric version

Outlook	Temperature	Humidity	Wind	PlayTennis
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	83	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

3.5 Experiments and Results

This section presents the results obtained with FOLD-R algorithm on some of the standard UCI datasets. To conduct the following experiments, we implemented the algorithm in Java. We used Prolog queries to process the background knowledge (the background knowledge is assumed to be represented as a standard Prolog program). For performing information gain computations and CWA generation of negative examples, we made use of the JPL library (Singleton and Dushin, 2003) which interfaces SWI-Prolog (Wielemaker et al., 2012) with Java. Our intention here is to investigate the quality of discovered rules both in terms of their accuracy and the degree to which they are consistent with the common sense understanding from the underlying concepts. To measure the accuracy, we implemented 10-fold cross-validation on each dataset and the mean of calculated accuracy is represented while the standard deviation for all the datasets were 5 percent or lower. At present, we are not greatly interested in the running time and/or space complexity of the algorithm: this will be subject of future research. All the learning tasks were preformed using a PC with Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz and 8.00 GB RAM and the execution time is just a matter of minutes if not seconds. The bottleneck is the function that sorts the numeric values to pick the best threshold and operator. There are solutions to get around this such as (Catlett, 1991).

Labor Relations: The data includes a set of contracts which depending on their features (16 features) are classified as good or bad contracts. The following set of clauses for a good contract are discovered by FOLD-R:

```
good_cont(X) :- wage_inc_f(X,A), A > 2, not ab0(X).
good_cont(X) :- holidays(X,A), A > 11.
good_cont(X) :- hplan_half(X), pension(X).
ab0(X) :- no_long_disability_help(X).
ab0(X) :- no_pension(X).
```

According to the first rule, a contract with 2 percent wage increase (default) is a good contract except when the employer does not contribute in a possible long-term disability and a pension plan. According to the second rule, a contract with holidays above 11 days is also good. And finally, if employer contributes in half of the health plan and entire pension plan, the contract is good.

Mushroom: This dataset includes descriptions of different species of mushrooms and their features which are used to classify whether they are poisonous or edible. The following set of clauses for a poisonous mushroom is discovered by FOLD:

```
poisonous(X) :- ring_type_none(X).
poisonous(X) :- spore_print_color_green(X).
poisonous(X) :- gill_size_narrow(X), not ab2(X).
poisonous(X) :- odor_foul(X).
ab0(X) :- population_clustered(X).
ab0(X) :- stalk_surface_below_ring_scaly(X).
ab1(X) :- stalk_shape_enlarging(X).
ab2(X) :- gill_spacing_crowded(X),not ab1(X).
ab2(X) :- odor_none(X), not ab0(X).
```

Note, the induced theory has nested exceptions. This nesting happens as a result of finding patterns for negative examples, which makes the FOLD algorithm perform more recursions until no covered negative example is left.

Table 3.2 compares the accuracy of FOLD-R algorithm against that of ALEPH (Srinivasan, 2001). The examples have been picked from well-known standard datasets for some of which ALEPH exhibits low test accuracy. In most cases, FOLD-R accuracy outperforms ALEPH. The experiments suggest when absence of a particular feature value plays a crucial role in classification, our algorithm shows a meaningful higher accuracy. This comes from the

Table 3.2: FOLD-R evaluation on UCI benchmarks

dataset	size	ALEPH accuracy(%)	FOLD-R accuracy(%)
Credit-au	690	82	83
Credit-j	125	53	81
Credit-g	1000	70.9	78
Iris	150	85.9	95
Ecoli	336	91	90
Bridges	108	89	90
Labor	57	89	94
Acute(1)	34	100	100
Acute(2)	34	100	100
Mushroom	7724	100	100

fact that the classical ILP algorithms only make use of existent information as opposed to *negation-as-failure* in which a decision is made based on the absence of information. As an example, in the credit-j dataset, our algorithm generates 4 rules with abnormality predicates. These rules cover positive examples which without abnormality predicates would have remained uncovered. However, in Bridges and Ecoli where no abnormality predicate is introduced by our algorithm, both ALEPH and FOLD-R end-up with almost the same accuracy.

Even in cases where no improvement over accuracy is achieved, our default theory approach leads to simpler and more intuitive rules. As an example, in case of Mushroom, other ILP systems, including ALEPH and FOIL, would produce 9 rules with 2 literals each in the body to cover all the positives, while our FOLD algorithm, produces 3 single-literal rules and 1 rule with 2 literals in which the second literal takes care of the exceptions.

3.6 Non Observation Learning Using FOLD

In usual machine learning setting of "Observation Predicate Learning" (OPL), examples and hypotheses define the same predicate. In contrast, non-OPL setting allows to have examples other than the ground target predicate. Non-OPL setting is natural for many

problems (Muggleton and Bryant, 2000). Therefore, a natural extension of FOLD would be to include non-OPL setting. Intuitively, non-OPL requires to obtain how each non-target example impacts the correct hypothesis in terms of target ground atoms. The following example shows how a non-target ground predicate could be expressed in terms of positive and negative examples of the target predicate.

Example 3.5. *Consider the following Background knowledge. Given the positive example set* $E^+ = \{p(a), r(c)\}, E^- = \{p(d)\}$*, we want to learn the target* $r(X)$*.*

```
(1) p(X) :- s(X), not r(X).      (3) q(a,b).
(2) s(X) :- q(X,Y), r(Y).        (4) s(d).
```

Since $B \cup H$ *must imply* $p(a)$*, from rule (1) we get* $s(a)$ *must hold and* $r(a)$ *should not. For* $s(a)$ *to hold, from rule (2) we get* $q(a, Y), r(Y)$ *must hold. Such* Y *indeed exists from fact (3). Therefore,* $r(b)$ *must hold too.* $p(a)$ *requires* $r(b)$ *and not* $r(a)$*. Therefore,* $p(a)$ *can be replaced by new examples, i.e.,* $r(b)$ *a new positive example, and* $r(a)$*, a new negative example. The impact of* $p(d)$ *as a negative example is to force* $r(d)$ *not to hold, because, from (4) we get* $s(d)$ *holds, therefore,* $r(d)$ *must not. Hence,* $r(d)$ *is a new negative example and replaces* $p(d)$*.*

The computation performed in Example 3.5 to replace non-target examples with target examples is realized using abduction in a goal-directed answer set programming system called s(ASP) (Marple et al., 2017; Gupta, 2017). The s(ASP) system takes an answer set program P and a query goal Q as inputs and enumerates all answer sets that contain the propositions/predicates in Q. This enumeration employs co-inductive SLD resolution to systematically compute elements of the greatest fixed point (GFP) of a program via backtracking. The advantage of s(ASP) over other answer set solvers is that it would lift the restriction that answer set programs must be finitely groundable. In order to process a query Q, s(ASP) would produce a set called the "partial answer set" containing the elements that

are necessary to establish Q. The s(ASP) system also allows a query to run abductively, by first defining a set of predicates as abducible. By doing so, if success of a query Q depends on assuming a fact that belongs to the set of abducibles, Q abductively succeeds and the abducibles are added to the set of partial answer set associated with Q.

Algorithm 4 shows the required steps in order to solve a non-OPL ILP problem using FOLD. In case of Example 3.5, $p(a)$ is a non-target example. By running s(ASP) and defining

Algorithm 4 Non-OPL Version of FOLD Algorithm

Input: $target, B, E^+, E^-$
Output: Hypothesis H

1: $abduced^+, abduced^- = \{\ \}$
2: Let Q be the query: $? - E^+,\ not\ E^-$
3: Run Q on s(ASP) $\langle B, \#abducibles = \{target\}\rangle$ ▷ Run s(ASP) with B as input
4: Let P = partial answer set associated with Q
5: **for each** $p \in P$ $s.tpred(p)\ == target$ **do**
6: **if** $sign(p)\ == +$ **then**
7: $abduced^+ \leftarrow abduced^+ \cup \{p\}$
8: **else**
9: $abduced^- \leftarrow abduced^- \cup \{p\}$
10: **end if**
11: **end for**
12: $E^+ \leftarrow E^+ \cup\ abduced^+$
13: $E^- \leftarrow E^- \cup\ abduced^-$
14: Run $FOLD\langle B, E^+, E^-, target\rangle$

`#abducible r(X)`, the following partial answer set is produced by s(ASP) on the following query: `?- p(a).`

```
{p(a), q(a,b), r(b), s(a), not r(a)}
```

$r(b)$ and $r(a)$ are added to the set of positive and negative examples, respectively. It should be noted that the above set of predicates are relevant to establish the query `?- p(a)`. In practice, this is a small subset of the original stable model. The fact that s(ASP) does not ground the answer set program, makes this approach scalable comparing to SAT based answer set solvers.

3.7 Related Work

Sakama in (Sakama, 2005) discusses the necessity of having a non-monotonic language bias to perform induction for default reasoning. It surveys some of the proposals directly adapted from ILP, like *inverse resolution* (Muggleton and Buntine, 1988) and *inverse entailment* (Muggleton, 1995a) then he explains why these are not applicable to the non-monotonic logic programs. Sakama then introduces an algorithm to induce rules from *answer sets* which generalizes a rule from specific grounded rules in a bottom-up fashion. His approach in some cases yields premature generalizations that produces redundant negative literals in the body of the rule and therefore over-fitted to the "training data". The following example illustrates what Sakama's algorithm would produce:

Example 3.6.

$\mathcal{B}:$ $bird(X) \leftarrow penguin(X).$

$bird(tweety).$ $bird(et).$

$bear(teddy).$ $crippled(et).$

$cat(kitty).$ $penguin(polly).$

$\mathcal{E}^+:$ $fly(tweety).$

and the algorithm outputs the following rule:

```
fly(X) :- bird(X), not cat(X), not penguin(X), not bear(X), not crippled(X).
```

in which some of the literals including not cat(X) and not bear(X) are redundant.

Additionally, since ASP systems have to ground the predicates to produce the answer set, introducing numeric data in background knowledge and also in the language bias is prohibited. Similarly, (Inoue and Kudoh, 1997) proposes a bottom-up algorithm in two phases: First, producing monotonic rules by a Horn ILP, then specializing them by introducing negated literals to the body of the rule.

In a different line of research (Dimopoulos and Kakas, 1995), describes an algorithm to learn exception using the patterns in the negative examples. However, they don't make any use of NAF as the core notion of reasoning in the absence of complete information and instead their algorithm learns a hierarchical logic program including classical negation in which the order of rules prioritize their application and therefore it's not naturally compatible with standard Prolog.

The idea of swapping positive and negative examples to learn patterns from negative examples has first been discussed in (Srinivasan et al., 1996) where a bottom-up ILP algorithm is proposed to specialize a clause after it has already been generalized and still covers negative examples. In contrast, we believe our FOLD algorithm with a top-down approach is a better fit thanks to its support for numeric features and better scalability, lack of both are inherent problems in bottom-up ILP methods.

ALEPH (Srinivasan, 2001) is one of the most widely used ILP systems that uses a bottom-up generalization search to induce theories that covers the maximum possible positive examples. However, since the induced theory might be overly generalized, there is an option to refine the theory by introducing abnormality predicates that rule out negative examples by specializing an overly generalized rule. This specialization step is manual and unlike our algorithm, no automation is offered by ALEPH. Also, ALEPH does not support numeric features.

One of the advantages of our FOLD-R algorithm over the existing systems is the ability to handle non-monotonic background knowledge. The conventional ILP systems use standard Prolog to handle the background whereas, FOLD-R once integrated with a top down *answer set programming* system like s(ASP) (Marple and Gupta, 2012), queries the background knowledge instead of producing the entire answer set and hence is scalable and applicable to a non-monotonic background knowledge. Further improvement on the accuracy of model predictions using boosting techniques and providing the justification when ensemble methods

are performed, is subject to more research. Future work also includes applying our algorithms to real world problems with large datasets.

CHAPTER 4

INDUCTIVE LEARNING OF ASP PROGRAMS WITH MULTIPLE STABLE MODELS

4.1 Overview

ILP learns theories in the form of Horn clause logic programs. Extending Horn clauses with negation as failure (NAF) results in more powerful applications becoming possible as inferences can be made even in absence of information. This extension of Horn clauses with NAF where the meaning is computed using the stable model semantics (Gelfond and Lifschitz, 1988)—called Answer Set Programming [1]—has many powerful applications. Generalizing ILP to learning answer set programs also makes ILP more powerful. For a complete discussion on the necessity of NAF in ILP, we refer the reader to (Sakama, 2005).

Once NAF semantics is allowed into ILP systems, they should be able to deal with multiple stable models which arise due to presence of mutually recursive rules involving negation (called *even cycles*) (Gelfond and Lifschitz, 1988) such as:

```
p :- not q.
q :- not p.
```

XHAIL (Ray, 2009), ASPAL (Corapi et al., 2012), ILASP (Law et al., 2014) are among the recently emerged systems capable of learning non-monotonic logic programs. However, they all resort to an exhaustive search for the hypothesis. The exhaustive search is not scalable on practical datasets. For instance, (all versions of) ILASP training procedure times-out after couple of hours on "Moral Reasoner" a dataset from the UCI repository. This is a small dataset containing roughly 200 examples and 50 candidate predicates in language bias.

[1] We use the term answer set programming in a generic sense to refer to normal logic programs, i.e., logic programs extended with NAF, whose semantics is given in terms of stable models (Gelfond and Kahl, 2014).

In contrast, traditional ILP systems (that only learn Horn clauses), use heuristics to guide their search. Use of heuristics allows them to avoid an exhaustive search. These systems usually start with the most general clauses and then specialize them. They are better suited for large-scale data-sets with noise, since the search can be easily guided by heuristics. FOIL (Quinlan, 1990a) is a representative of such algorithms. However, handling negation in FOIL is somewhat problematic as we discuss in (Shakerin et al., 2017). Also, FOIL cannot handle background knowledge with multiple stable models, nor it can induce answer set programs.

In chapter 3 we presented the FOLD a algorithm (Shakerin et al., 2017) to automate inductive learning of default theories represented as stratified answer set programs. FOLD (First Order Learner of Default rules) extends the FOIL algorithm and is able to learn answer set programs that represent the underlying knowledge very succinctly. However, FOLD is only limited to dealing with stratified answer set programs, i.e., mutually recursive rules through negation are not allowed in the background knowledge or the hypothesis. Thus, FOLD is incapable of handling cases where the background knowledge or the hypothesis admits multiple stable models. In this chapter, we extend the FOLD algorithm to allow both the background knowledge and the hypothesis to have multiple stable models. The extended FOLD algorithm—called the XFOLD algorithm—is much more general than previously proposed methods.

4.2 The XFOLD Algorithm

In this section we extend our FOLD algorithm to learn normal logic programs that potentially have multiple stable models. The significance of Answer Set Programming paradigm is that it provides a declarative semantics under which each stable model is associated with one (alternative) solution to the problem described by the program. Typical problems of this kind are combinatorial problems, e.g., graph coloring and N-queens. In graph coloring, one should find different ways of coloring nodes of a graph without coloring two nodes connected

by an edge with the same color. N-queen is the problem of placing N queens in a chessboard of size $N \times N$ so that no two queens attack each other.

In order to inductively learn such programs, the ILP problem definition needs to be revisited. In the new scenario, positive examples $e \in E^+$, may not hold in every model. Therefore, the ILP problem described in the background section would only allow learning of predicates that hold in all answer sets. This is too restrictive. Brave induction (Sakama and Inoue, 2009), in contrast, allows examples to hold only in some stable models of $B \cup H$. However, as stated in (Law et al., 2014), and we will show using examples, this is not enough when it comes to learning global constraints (i.e, rules with empty head)[2]. Learning global constraints is essential because certain combinations may have to be excluded from *all* answer sets.

When $B \cup H$ has multiple stable models, there will be some instances of target predicate that would hold in all, none, or some of the stable models. Brave induction is not able to express situations in which a predicate should hold in all or none of the stable models. An example is a graph in which node 1 is colored red. In such a case, none of node 1's neighbors should be colored red. If node 1 happens to have node 2 as a neighbor, brave induction is not able to express the fact that if the atom `red(1)` appears in any stable model of $B \cup H$, `red(2)` should not. In (Law et al., 2014), the authors propose a new paradigm called learning from partial answer sets that overcomes these limitations. We also adopt this paradigm in this work. Next, we present our XFOLD algorithm.

Definition 4.1. *A partial interpretation E is a pair $E = \langle E^{inc}, E^{exc} \rangle$ of sets of ground atoms called inclusions and exclusions, respectively. Let $A \in AS(B \cup H)$ denote a stable model of $B \cup H$. A extends $\langle E^{inc}, E^{exc} \rangle$ if and only if $(E^{inc} \subseteq A) \wedge (E^{exc} \cap A = \emptyset)$.*

[2]Recall that in answer set programming, a constraint is expressed as a headless rule of the form

```
:- B.
```

which states that `B` must be false. A headless rule is really a short-form of rules of the form (called odd loops over negation (Gelfond and Kahl, 2014)):

```
p :- B, not p.
```

Example 4.1. *Consider the following background knowledge about a group of friends some of whom are in conflict with others. The individuals in conflict will not attend a party together. Also, they cannot attend a party if they work at the time the party is held. We want our ILP algorithm to discover the rule(s) that will determine who will go to the party based on the set of partial interpretations provided.*

B :
```
conflict(X,Y) :- person(X), person(Y), conflict(Y,X).
works(X) :- person(X), not off(X).
off(X) :- person(X), not works(X).
person(p1).  person(p2). conflict(p1,p4).
person(p3).  person(p4).  person(p5). conflict(p2,p3).
```

Some of the partial interpretations are as follows:

The predicates g,w,o abbreviate goesToParty, works, off respectively:

$E_1 = \{\langle g(p1), g(p2), o(p1), o(p2), w(p3), o(p4), w(p5)\rangle, \langle g(p3), g(p4), g(p5)\rangle\}$

$E_2 = \{\langle g(p3), g(p4), g(p5), o(p1), o(p2), o(p3), o(p4), o(p5)\rangle, \langle g(p1), g(p2)\rangle\}$

$E_3 = \{\langle g(p1), g(p3), g(p5), o(p1), o(p2), o(p3), w(p4), o(p5)\rangle, \langle g(p2), g(p4)\rangle\}$

$E_4 = \{\langle g(p2), g(p5), g(p5), w(p1), o(p2), w(p3), w(p4), o(p5)\rangle, \langle g(p1), g(p3), g(p4)\rangle\}$

In the above example, each E_i for i = 1,2,3,4 is a partial interpretation and should be extended by at least one stable model of $B \cup H$ for a learned hypothesis H. For instance, let's consider the hypothesis $H_1 = \{$`goesToParty(X) :- off(X)`$\}$ for learning the target predicate `goesToParty(X)`. By plugging the background knowledge, the non-target predicates in E_1, and the hypothesis H_1 into an ASP solver (CLASP (Gebser et al., 2012) in our case), the stable model returned by the solver would contain the following:

$\{$`goesToParty(p1),goesToParty(p2),goesToParty(p4)`$\}$.

It does not extend E_1. Although, $E_1^{inc} \subseteq AS(B \cup H_1)$ but $AS(B \cup H_1) \cap E_1^{exc} \neq \emptyset$. It should be noted that non-target predicates are treated as background knowledge upon calling ASP solver to compute the stable model of $B \cup H$.

Definition 4.2. *An XFOLD problem is defined as a tuple* $P = \langle B, L, E^+, E^-, T \rangle$. *B is a answer set program with potentially multiple stable models called the background knowledge. L is the language-bias such that* $L = \langle M_h, M_b \rangle$, *where* M_h *(resp.* M_b*) are called the head (resp. body) mode declarations (Muggleton, 1995b).*

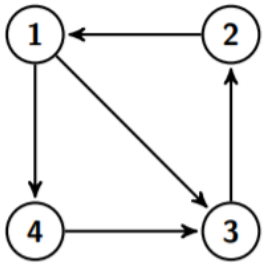
Each mode declaration $m_h \in M_h$ (resp. $m_b \in M_b$) is a literal whose abstracted arguments are either variable v or constant c. Type of a variable is a predicate defined in B. The domain of each constant should be defined separately. Hypothesis h is said to be compatible with a mode declaration m if each instance of variable in m is replaced by a variable, and every constant takes a value from the associated domain. The set of candidate predicates in the greedy search algorithm are selected from $M_b \cup M_h$.

XFOLD is extended with mode declaration to make sure that every clause generated is safe for the ASP solver CLASP as it needs to ground the program. To obtain a finite grounded program, CLASP must ensure that every variable is safe. A variable in *head* is *safe* if it occurs in a positive literal of body. XFOLD adds predicates required to ensure safety, but to keep our examples simple, we omit safety predicates. E^+ and E^- are sets of partial interpretations called positive and negative examples, respectively. $T \in M_h$ is the target predicate's name. Each XFOLD run learns a single target predicate. A hypothesis $h \in L$ is an inductive solution of T if and only if:

1. $\forall e^+ \in E^+ \exists A \in AS(B \cup H)$ such that A extends e^+

2. $\forall e^- \in E^- \nexists A \in AS(B \cup H)$ such that A extends e^-

The above definition adopted from (Law et al., 2014) subsumes brave and cautious induction semantics (Sakama and Inoue, 2009). Positive examples should be extended by at least one stable model of $B \cup H$ (brave induction). In contrast, no stable model of $B \cup H$ extends negative examples (cautious induction). The *generate and test* problems such as



– Positive examples:

$E_1^+ = \{\langle r(1), b(2), g(3), b(4)\rangle, \langle not\ b(1), not\ g(1), not\ r(2), not\ g(2), not\ r(3), not\ b(3), not\ r(4), not\ g(4)\rangle\}$

$E_2^+ = \{\langle b(1), r(2), g(3), r(4)\rangle, \langle not\ r(1), not\ g(1), not\ b(2), not\ g(2), not\ r(3), not\ b(3), not\ b(4), not\ g(4)\rangle\}$

– Negative examples:

$E_1^- = \{\langle r(1)\rangle, \langle not\ r(2)\rangle\}$ $E_2^- = \{\langle r(1)\rangle, \langle not\ r(3)\rangle\}$

Figure 4.1: Partial interpretations as examples in graph coloring problem

N-queen and graph coloring could be induced using our XFOLD algorithm. It suffices to use positive examples for learning the *generate* part and negative examples for learning the *test* part.

Figure 4.1 represents the input to the XFOLD algorithm for learning an answer set program for graph coloring. Every positive example states if a node is colored red, then that node cannot be painted blue or green. Likewise for blue and green. However, this is not enough to learn the constraint that two nodes connected by an edge cannot have the same color. To learn this constraint, negative examples are needed. For instance, E_1^-, states that if any stable model of $B \cup H$ contains {`red(1)`}, in order not to extend E_1^-, it should contain {`not red(2)`} or equivalently, it should not contain {`red(2)`}. Intuitively, XFOLD is similar to FOLD and FOIL: To specialize a clause cl, for every positive example $e \in E^+$, the background knowledge B, all non-target predicates in e^{inc} and cl are passed to the ASP solver as inputs. The resulting answer set is compared with the target predicates in e^{inc} and e^{exc} to compute a partial score. Next, by summing up all partial scores, total score of that clause is computed. Among all candidate clauses, the one with highest total score is selected. Once for all $e \in E^+$ no target predicate in e^{exc} is covered, the internal loop finishes and the discovered rule(s) are added to the learned theory. Just like FOLD, if no literal with positive score exists, swapping occurs on each remaining partial interpretation and the XFOLD algorithm is recursively called. In this case, instead of introducing abnormality

Algorithm 5 The XFOLD Algorithm

Input: $target,\ B,\ \{e = (e^{inc}, e^{exc}) | e \in E^+)\}$
Output: Hypothesis H
  function SPECIALIZE(cl, B, E^+) ▷ Other functions remain unchanged as in FOLD
    while $\exists e \in E^+$ *such that* $e^{exc}! = \emptyset$ **do**
      for each $c \in \rho(cl)$ **do** ▷ FOIL inner loop (refinement)
        for each $e_i \in E^+$ **do**
          compute $partial\_score[i][c]$ ▷ partial score for each clause
        end for
        $total\_score[c] = \sum_{e_i \in E^+} partial\_score[i][c]$
      end for
      Let c_best, max_score, be the clause with the highest score and its associated score
      if $max\_score > 0$ **then**
        cl $\leftarrow c\_best$
        $H \leftarrow H \cup \{cl\}$
      else
        $E\_swapped^+ = \text{Swap}(E^+)$
        XFold($B, E\_swapped^+, -target$)
      end if
      update E^+
    end while
  end function

predicates, the negation symbol, "-", is prefixed to the current target predicate to indicate that the algorithm is now trying to learn the negation of concept being learned. It should also be noted that swapping examples is performed slightly differently due to the existence of partial interpretations. For each $e \in E^+$ the following operations are performed upon swapping:

1. $\forall t \in e^{inc}$, where t is an old target atom already covered and removed, t is restored
2. $\forall t \in e^{inc}$, where t is an old target atom, $-t$ is added to e^{exc}
3. $\forall t \in e^{exc}$, where t is an old target atom, $-t$ is added to e^{inc}
4. $T \leftarrow -T$. (Target predicate T now becomes its negation, -T)

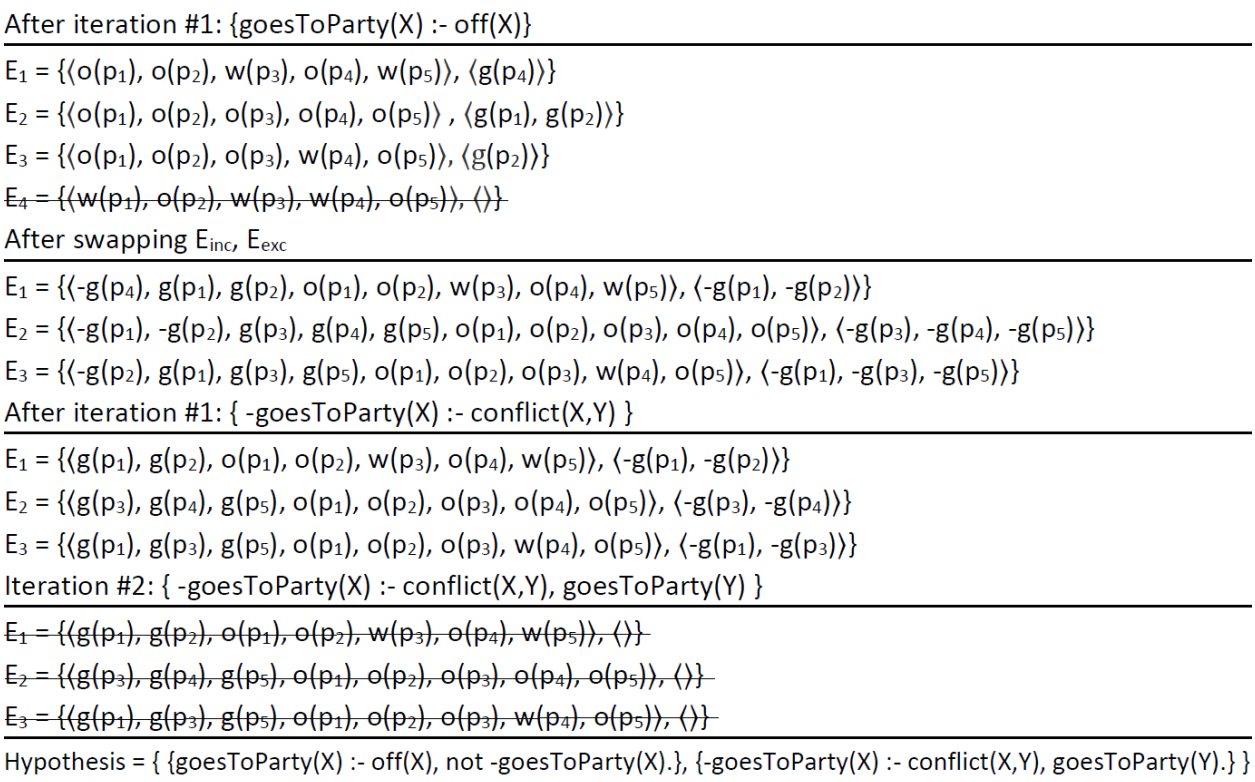


Figure 4.2: Trace of XFOLD execution on the Party Example

Figure 4.2 shows execution of XFOLD on Example 4.1. At the end of first iteration, the predicate `off(X)` gets the highest score. E_4 will be removed as it is already covered by the current hypothesis. In the second iteration, all candidate literals fail to get a positive score. Therefore, swapping of positive and negative examples occurs and algorithm tries to learn the predicate `-goesToParty(X)`. Since the new target predicate is `-goesToParty(X)`, all ground atoms of `goesToParty` in E^{inc} are restored back. The old target atoms in E^{exc} are transformed to negated version and become members of E^{inc}. In Figure 4.2, after one iteration E_4 is removed because all target atoms in E^{inc} are already covered and targets atoms in E^{exc} are already excluded. After swapping, XFOLD is recursively called to learn `-goesToParty`. After 2 iterations, all examples are covered and the algorithm terminates.

In Example 4.1, we haven't introduced any explicit negative example. Nevertheless, the algorithm was able to successfully find the cases in which the original target predicate does

not hold (via learning `-goesToParty(X)` predicate). In general, it is not always feasible for the algorithm to figure out prohibited patterns without getting to see a very large number of positive examples.

4.3 Application: Combinatorial Problems

A well-known methodology for declarative problem solving is the *generate and test* methodology, whereby possible solutions to a problem are generated first, and then non-solutions are eliminated by testing. In Answer Set Programming, the *generate* part is encoded by enumerating the possibilities by introducing even cycles. The *test* part is realized by having constraints that would eliminate answer sets that violate the test conditions. ASP syntax allows rules of the form $l\{h_1, ..., h_k\}u$ such that $0 \leq l \leq u \leq k$ and $\forall i \in [1, k]$, $h_i \in L$, where L is the language bias. This syntactic sugar for combination of even cycles and constraints is called *choice rule* in the literature (Gelfond and Kahl, 2014).

ILASP (Law et al., 2014) directly searches for choice rules by including them in the search space. XFOLD, on the other hand, performs the search based on θ-subsumption (Plotkin, 1971) and hence disallows search for choice rule hypotheses. Instead, it directly learns even cycles as well as constraints. This is advantageous as it allows for more sophisticated and flexible language bias.

It turns out that inducing the *generate* part in a combinatorial problem such as graph-coloring requires an extra step compared to the FOLD algorithm. For instance, `red(X)` predicate has the following clause:

```
red(X):- not blue(X), not green(X).
```

To enable XFOLD to induce such a rule, we adopted the "Mathews Correlation Coefficient" (MCC) (Zeng et al., 2014) measure to perform the task of feature selection. MCC is calculated as: $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

This measure takes into account all the four terms TP (true positive), TN (true negative), FP (false positive) and FN (false negative) in the confusion matrix and is able to fairly assess the quality of classification even when the ratio of positive tuples to the negative tuples is not close to 1. The MCC values range from -1 to +1. A coefficient of +1 represents a perfect classification, 0 represents a classification that is no better than a random classifier, and -1 indicates total disagreement between the predicted and the actual labels. MCC cannot replace XFOLD heuristic score, i.e, *information gain*, because the latter tries to maximize the coverage of positive examples, while the former only maximally discriminates between the positives and negatives. Nevertheless, for the purpose of feature extraction among the negated literals which are disallowed in XFOLD algorithm, MCC can be applied quite effectively. For that matter, before running XFOLD algorithm, the MCC score of all candidate literals are computed. If a predicate scores "close" to +1, the predicate itself is added to the language bias. If it scores "close" to -1, its negation is added to the language bias. For example, in case of learning `red(X)`, after running the feature extraction on the graph given in Figure 4.1, XFOLD computes the scores -0.7, -0.5 for `green(X)` and `blue(X)`, respectively. Therefore, {`not green(X),not blue(X)`} are appended to the list of candidate predicates. Now, after running the XFOLD algorithm, after two iterations of the inner loop, it would produce the following rule:

```
red(X) :- not green(X), not blue(X).
```

Corresponding rules for `green(X)` and `blue(X)` are learned in a similar manner. This essentially takes care of the *generate* part of the combinatorial algorithm. In order to learn the *test* part for graph coloring, we need the negative examples shown in Figure 4.1. It should be noted that in order to learn a constraint, we first learn a new target predicate which is the negation of the original one. Then we shift the negated predicate from the head to the body inverting its sign in the process. That is, we first learn a clause of the form {`-T :-` b_1`,`

b_2 ... b_n.} which is then transformed into the following constraint: {:- b_1, b_2 ... b_n, T.} Thus, the following steps should be taken to learn constraints from negative examples:

1. Add rule(s) induced for *generate* part to B.
2. $\forall e^+ \in E^+, e^- \in E^-$, if $e^-_{inc} \subseteq e^+_{inc}$:
   - **if** e^-_{exc} is of the form (not $p(V_1, ...V_m)$) **then** $e^+_{inc} \leftarrow e^+_{inc} \cup \{-p(V_1, ...V_m)\}$
   - **else** $e^+_{exc} \leftarrow e^+_{exc} \cup \{-p(V_1, ...V_m)\}$
3. compute the contrapositive form of the rule(s) learned in *generate* part and remove the body predicates from the list of candidate predicates
4. run XFOLD to learn p
5. shift -p from the head to the body for each rule returned by XFOLD

The contrapositive form of a clause is computed by negating the head and applying the De Morgan's law to the body. The resulting disjunctions are resolved by separating them into new clauses. For instance, the contrapositive of {red(X) :- not green(X), not blue(X)} is obtained as follows:
{-red(X) :- green(X)},{-red(X) :- blue(X)}. Without step 3, XFOLD would learn these trivial clauses. However, as soon as those trivial choices are removed from search space, XFOLD algorithm comes up with the next best hypothesis which is as follows:

```
-red(X) :- edge(X,Y), red(Y).
```

Shifting -red(X) to the body yields the following constraint:

```
:- red(X),edge(X,Y),red(Y).
```

In graph coloring problem, M_h = {red(X), green(X), blue(X)}. Once similar examples for green(X) and blue(X) are provided, XFOLD is able to learn the complete solution as shown below:

Algorithm 3 The generate and Test Version of XFOLD algorithm

Input: $L = \langle M_h, M_b \rangle, B, E^+, E^-$
Output: Hypothesis H

% - **Induction of "generate" part** - %
Initialize $H \leftarrow \emptyset$
for each $t \in M_h$ **do**
    for each $l \in M_b$ **do**
        $L \leftarrow L \cup \{not\ l\}$ if MCC of the clause {`t :- not l`} is close to +1
    end for
    $h_t \leftarrow$ XFOLD $\langle B, L, E^+_{inc}, E^+_{exc}, t \rangle$
    $H \leftarrow H \cup h_t$
end for
$B \leftarrow B \cup H$
% - **Induction of "test" part** - %
for each $t \in M_h$ **do**
    for each $e^+ \in E^+, e^- \in E^-$ **do**
        if $e^-_{inc} \subseteq e^+_{inc}$ **then**
            if e^-_{exc} is of the form `not t(`$V_1, ..., V_m$`)` **then**
                $e^+_{inc} \leftarrow e^+_{inc} \cup \{$`-t(`$V_1, ..., V_m$`)`$\}$
            else
                $e^+_{exc} \leftarrow e^+_{exc} \cup \{$`-t(`$V_1, ..., V_m$`)`$\}$
            end if
        end if
    end for
end for
compute the contrapositive form for each $h \in H$ in *generate* part and remove the body predicates from the list of candidate predicates L
for each $t \in M_h$ **do**
    $h_t \leftarrow$ XFOLD $\langle B, L, E^+_{inc}, E^+_{exc}, -t \rangle$
    shift -t from the head to the body to get a constraint $\hat{h}_t$
    $H \leftarrow H \cup \{\hat{h}_t\}$
end for

```
red(X) :- not green(X), not blue(X).

green(X) :- not blue(X), not red(X).

blue(X) :- not green(X), not red(X).

:- red(X), edge(X,Y), red(Y).

:- blue(X), edge(X,Y), blue(Y).

:- green(X), edge(X,Y), green(Y).
```

Algorithm 3 shows how XFOLD induces a *generate and test* hypothesis.

Example 4.2. *Learning an answer set program for the 4-queen problem. The following items are assumed: Background knowledge B including predicates describing a 4×4 board, rules describing different ways through which two queens attack each other and examples of the following form:*

B: `attack_r(`R_1`,`C_1`,`R_2`,`C_2`):-q(`R_1`,`C_1`),q(`R_2`,`C_2`),`$C_1 != C_2$`, `$R_1 = R_2$`.`

`attack_c(`R_1`,`C_1`,`R_2`,`C_2`):-q(`R_1`,`C_1`),q(`R_2`,`C_2`),`$R_1 != R_2$`, `$C_1 = C_2$`.`

`attack_d(`R_1`,`C_1`,`R_2`,`C_2`):-q(`R_1`,`C_1`),q(`R_2`,`C_2`),`$R_1 != R_2$`,`$R_1 - C_1 = R_2 - C_2$`.`

`attack_d(`R_1`,`C_1`,`R_2`,`C_2`):-q(`R_1`,`C_1`),q(`R_2`,`C_2`),`$R_1 != R_2$`,`$R_1 + C_1 = R_2 + C_2$`.`

E: $E_1^+ = \{\langle q(2,1), q(4,2), q(1,3), q(3,4)\rangle, \langle q(1,1), q(1,2), ..., q(4,4)\rangle\}$

...

$E_1^- = \{\langle q(2,1)\rangle, \langle not\ q(2,2)\rangle\}$

$E_2^- = \{\langle q(2,1)\rangle, \langle not\ q(2,3)\rangle\}$

$E_3^- = \{\langle q(4,2)\rangle, \langle not\ q(1,2)\rangle\}$

$E_4^- = \{\langle q(4,2)\rangle, \langle not\ q(2,3)\rangle\}$

As far as the *generate* part is concerned, XFOLD algorithm learns the following program:

```
q(X,Y) :- not -q(X,Y).
-q(X,Y) :- not q(X,Y).
```

The predicate -q(X,Y) is introduced by XFOLD algorithm as a result of swapping the examples and calling itself recursively. After computing the contrapositive form, `q(X,Y)`, `-q(X,Y)` are removed from the list of candidate predicates. Then based on the examples provided in Example 4.2, XFOLD would learn the following rules:

```
-q(V1,V2) :- attack_r(V1,V2,V3,V4).
-q(V1,V2) :- attack_c(V1,V2,V3,V4).
-q(V1,V2) :- attack_d(V1,V2,V3,V4).
```

After shifting the predicate `-q(`V_1`,`V_2`)` to the body, we get the following constraint:

```
:- q(V1,V2), attack_r(V1,V2,V3,V4).
:- q(V1,V2), attack_c(V1,V2,V3,V4).
:- q(V1,V2), attack_d(V1,V2,V3,V4).
```

It should be noted that, since XFOLD is a sequential covering algorithm like FOIL, it takes three iterations before it can cover all examples which in turn becomes three constraints as shown above.

4.4 Experiments and Results

Table 3.2 reports the classification accuracy using 10-fold cross-validation and running time measurements of Aleph, FOLD and XFOLD on a number of UCI datasets (Lichman, 2013a) and combinatorial problems discussed in this chapter. In (Shakerin et al., 2017) we compare our FOLD algorithm with Aleph which is a state-of-the-art ILP system. However, Aleph (Srinivasan, 2001) does not support multiple stable model ILP. Therefore, we can only compare our results with that of ILASP. In case of UCI datasets, the "Size" column denotes the number of data samples, whereas, in graph-coloring (N-queen) it denotes the number of nodes(board size) respectively. We have also examined the application of statistical feature selection on the performance of our XFOLD algorithm. We report a significant improvement due to the application of a scalable feature-selection method, i.e., xgboost, prior to invoking the learning algorithm. Exclusion of low ranked features and the use of negation-as-failure results in a significant improvement over the accuracy of learned hypotheses.

"Extreme Gradient Boosting" (xgboost) (Chen and Guestrin, 2016) is a scalable and powerful ensemble classifier based on decision trees that provides a feature importance score. Since, in ILP we deal with propositions, it makes sense to discretize numeric features first using MDL method (Fayyad and Irani, 1993a). In this method, for each numeric feature categories are defined such that the overall information gain is maximized.

Table 4.1: XFold Evaluation on UCI benchmarks and Combinatorial Problems

		Accuracy (%)				Running Time (s)	
Dataset	Size	Aleph	Fold	XFold	ILASP	XFold	ILASP
breast-cancer	286	70	82	88	—	4.1	timed-out
moral	202	96	96	100	—	4.8	timed-out
diabetes	768	73	86	89	—	27.2	timed-out
graph-coloring	4	—	—	100	100	8	4.5
graph-coloring	8	—	—	100	100	8.9	3.5
N-queen	4×4	—	—	100	100	9.5	5
N-queen	8×8	—	—	100	100	9.9	6.2

Next, a dataset that now contains only categorical features is propositionalized. That is, every value belonging to the domain of a categorical feature turns into a new binary feature. This is called *one hot encoding*. One hot encoding makes the feature selection more fine grained. This is because, in this technique instead of measuring the contribution of a feature as a whole, the importance of every value from the domain of that feature is measured. Then the data set is fed into xgboost which ranks each binary feature based on its importance in the classification. From the xgboost's output, the M lower ranked features are filtered out of the XFOLD language bias. The optimal M should be computed via cross-validation.

In small problems such as graph coloring, ILASP slightly outperforms our XFOLD algorithm due to embedding the learning algorithm in the ASP solver engine. In a larger data set such as Moral reasoner with 202 examples and 50 predicates, there are potentially 3^{50} different hypotheses to choose from. This is because, for each predicate it can either be included positively, included negatively or excluded. In this case, ILASP times out after couple of hours.

4.5 Related Work

A survey of extending Horn clause based ILP to non-monotonic logics can be found in (Sakama, 2005). "Stable ILP" (Seitzer, 1997) was the first effort to explore the expressiveness of background knowledge with multiple stable models. In (Sakama, 2005), Sakama introduces

algorithms to induce a *categorical* logic program[3] given the answer set of the background knowledge and *either* positive *or* negative examples. Essentially, given a single answer set, Sakama tries to induce a program that has that answer set as a stable model. In (Sakama and Inoue, 2009), Sakama and Inoue extend their work to learn from multiple answer sets. They introduce *brave* induction, where the learned hypothesis H is such that *some* of the answer sets of $B \cup H$ cover the positive examples. The limitation of this work is that it accepts only one positive example as a conjunction of atoms. It does not take into account negative examples at all. Cautious induction, the counterpart of brave induction, is also too restricted as it can only induce atoms in the intersection of all stable models. Thus, neither brave induction nor cautious induction are able to express situations where something should hold in all or none of the stable models. An example of this limitation arises in the graph coloring problem where the following should hold in all answer sets: no two neighboring nodes in a graph should be painted the same color.

ASPAL (Corapi et al., 2012) is the first ILP system to learn answer set programs by encoding ILP problems as ASP programs and having an ASP solver find the hypothesis. Its successor ILASP (Law et al., 2014), is a pioneering ILP system capable of inducing hypotheses expressed as answer set programs too. ILASP defines a framework that subsumes brave/cautious induction and allows much broader class of problems relating to learning answer set programs to be handled by ILP. However, the algorithm exhaustively searches the space of possible clauses to find one that is consistent with all examples and background knowledge. The Exhaustive search is a weaknesses that limits the applicability of ILASP to many useful situations. Our research presented in this chapter does not suffer from this issue.

XHAIL (Ray, 2009) is another ILP system capable of learning non-monotonic logic programs. It heavily incorporates abductive logic programming to search for hypotheses. It

[3]A categorical logic program is an answer set program with at most one stable model.

uses a similar language-bias as ILASP does, and thus suffers from the limitations similar to ILASP. It also does not support the notion of inducing answer set programs from partial answer sets.

CHAPTER 5

INDUCTION OF NON-MONOTONIC LOGIC PROGRAMS TO EXPLAIN MACHINE LEARNING MODELS

5.1 Overview

The ILP learning problem can be regarded as a search problem for a set of clauses that deduce the training examples. The search is performed either top down or bottom-up. A bottom-up approach builds most-specific clauses from the training examples and searches the hypothesis space by using generalization. This approach is not applicable to large-scale datasets, nor it can incorporate *Negation-As-Failure* into the hypotheses. A survey of bottom-up ILP systems and their shortcomings can be found at (Sakama, 2005). In contrast, top-down approach starts with the most general clauses and then specializes them. A top-down algorithm guided by heuristics is better suited for large-scale and/or noisy datasets (Zeng et al., 2014).

The FOIL algorithm by Quinlan (Quinlan, 1990b) is a popular top-down algorithm. FOIL uses heuristics from information theory called *weighted information gain*. The use of a greedy heuristic allows FOIL to run much faster than bottom-up approaches and scale up much better. For instance, the QuickFOIL system (Zeng et al., 2014) can deal with millions of training examples in a reasonable time. However, scalability comes at the expense of losing accuracy if the algorithm is stuck in local optima and/or when the number of examples is insufficient. The former is an inherent problem in hill climbing search and the latter is due to the shrinking of examples during clause specialization. Also, elimination of already covered examples from the training set (to guarantee the termination of FOIL) causes a similar impact on the quality of heuristic search for the best clause. Therefore, the predicates picked-up by FOIL are not always globally optimal with respect to the concept being learned. Based on our research, we believe that a successful ILP algorithm must satisfy the following criteria:

- It must employ heuristic-based search for clauses for the sake of scalability.
- It should be able to figure out relevant features, regardless of the number of current training examples.
- It should be able to learn from incomplete data, as well as be able to distinguish between noise and exceptions.

Unlike top-down ILP algorithms, statistical machine learning methods are bound to find the relevant features because they optimize an objective function with respect to global constraints. This results in models that are inherently complex and cannot explain what features account for a classification decision on any given data sample.

Recently, some solutions have been proposed by researchers to explain black-box classifiers' predictions locally. LIME (Ribeiro et al., 2016) is a novel model-agnostic system that explains the classification decisions made by any classifier on any given data sample. The idea comes from the fact that explaining classifier's behavior in a local region around any data turns out to be easier than explaining its global behavior. Each local explanation is a set of feature-value pairs that would determine what features and how strongly each feature, relative to other features, contributes to the classification decision.

In order to capture model's global behavior, we propose an algorithm called LIME-FOLD, to learn concise logic programs from a transformed data set that is generated by storing the explanations provided by LIME. The LIME system takes as input a black-box model (such as a Neural Network, Random Forest, etc.) and a data sample. For any given data sample, it outputs a list of (weighted) features that contribute most to the classification decision. By repeating the same process for all training samples, we can generate a transformed version of the original data set that only contains the relevant features for each data sample.

The LIME-FOLD algorithm learns a non-monotonic logic program from the transformed data set. This logic program explains the global behavior of the model. Our experiments

on 10 UCI standard benchmark suggests that the hypotheses generated by LIME-FOLD algorithm are very concise and outperform the baseline ALEPH system (Srinivasan, 2001). It also outperforms ALEPH once ALEPH is given the transformed dataset (i.e., ALEPH is extended with the LIME technique). Performance is measured in terms of classification evaluation scores, number of generated clauses and running time.

Although LIME is model-agnostic, in this research we incorporate the XGBoost algorithm to train our statistical models. XGBoost (Chen and Guestrin, 2016) is a scalable tree boosting machine learning algorithm that is widely used by data scientists to achieve state-of-the-art results on many challenges. In essence, the hypotheses (a nonmonotonic logic program) that our LIME-FOLD algorithm induces, explain the behavior of XGBoost models.

This chapter makes the following novel contribution: We present a new ILP algorithm capable of learning non-monotonic logic programs from local explanations of boosted tree models provided by LIME. We call this new algorithm LIME-FOLD. The LIME-FOLD algorithm is a scalable heuristic-based algorithm that explains the behavior of boosted tree models globally and outperforms ALEPH in terms of classification evalutation metrics as well as in providing more concise explanations measured in terms of number of clauses induced.

5.2 The LIME Technique

LIME (Ribeiro et al., 2016) is a novel technique that finds easy to understand explanations for the predictions of any complex black-box classifier in a faithful manner. LIME constructs a linear model by sampling N instances around any given data sample x. Every instance x' represents a perturbed version of x where perturbations are realized by sampling uniformly at random for each feature of x. LIME stores the classifier decision $f(x')$ and the kernel $\pi(x, x')$. The π function measures how similar the original and perturbed sample are and it is then used as the associated weight of x' in fitting a *locally weighted linear regression*

(LWR) curve around x. The K greatest learned weights of this linear model are interpreted as top K contributing features into the decision made by the black-box classifier. Algorithm 6 illustrates how a locally linear model is created around x to explain a classifier's decision.

Algorithm 6 Linear Model Generation by LIME

Input: f : Classifier
Input: N : Number of samples, K : length of explanation,
Input: x : sample to explain, π : similarity kernel
Output: w : fitted curve's weights

```
Z ← {}
for i ∈ {1, 2, 3, ..., N} do
    // x'_i is generated by perturbing features of x
    x'_i ← sample_around(x)
    Z ← Z ∪ ⟨x'_i, f(x'_i), π(x'_i, x)⟩
end for
// Fit a line to (weighted) points in Z
w ← LWR(Z, K)
return w
```

The interpretation language should be understandable by humans. Therefore, LIME requires the user to provide some interpretation language as well. In case of tabular data, it boils down to specifying the valid range of each table column. In particular, if the data column is a numeric variable (as opposed to categorical), the user must specify the intervals or a discretization strategy to allow LIME to create intervals that are used later on to explain the classification decision.

Example 5.1. *The UCI heart dataset contains features such as patient's blood pressure, chest pain, thallium test results, number of major vessels blocked, etc. The classification task is to predict whether the subject suffers from heart disease or not. Figure 5.1 shows how LIME would explain a model's prediction over a data sample.*

In this example, LIME is called to explain why the model predicts heart disease. In response, LIME returns the top features along with their importance weight. According to LIME, the model predicts "heart disease" because of high serum cholesterol level, and having

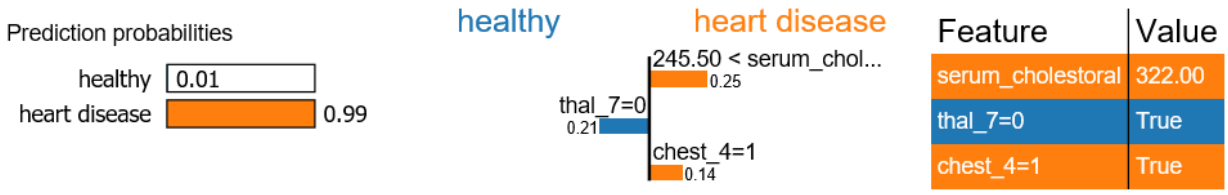


Figure 5.1: Top 3 Relevant Features in Patient Diagnosis According to LIME

a chest pain of type 4 (i.e., asymptomatic). In this dataset, chest pain level is a categorical variable with 4 different values.

The categorical variables should be *binarized* before a statistical model can be applied. Binarization is the process of transforming each categorical variable with domain of cardinality n, into n new binary features. The feature "thallium test" is a categorical feature too. However, in this case LIME reports that the feature "thal_7" which is a new feature that resulted from binarization and has the value "false", would have made the model predict "healthy". The value 7 for thallium test in this dataset indicates reversible defect which is a strong indication of heart disease. It should be noted that the feature "serum cholesterol" is discretized with respect to the training examples' label. Discretization aims to reduce the number of values a continuous variable takes by grouping them into intervals. Discretization method should maximize the interdependence between the variable values and the class labels. One of the most practiced methods for discretizing continuous data is the MDL method (Fayyad and Irani, 1993b) which uses mutual information to recursively define the best bins. In this research, we discretize all numeric features using the MDL method.

5.3 The LIME-FOLD Algorithm

In this section we introduce the LIME-FOLD algorithm by integrating FOLD and LIME. This yields a powerful ILP algorithm capable of learning very concise logic programs from a transformed dataset. The new algorithm outperforms FOLD and ALEPH (Srinivasan, 2001) which is a state-of-the-art ILP system.

There are two major issues with the *sequential covering* algorithms such as FOIL (and FOLD): 1) As number of examples decreases during specialization loop, probability of introducing an irrelevant predicate that accidentally splits a particular set of examples increases. 2) elimination of positive examples that are covered in previous iterations, impacts the precision of heuristic scoring. By filtering out the irrelevant features of each training example, the greedy clause search procedure is forced to pick up predicates only from a relevant subset of features to cover training examples. Relevant features for each training example is found by LIME once it is given an accurate classifier.

For instance in Figure 5.1, for a particular training example with 13 features, LIME returns only 3 as relevant to the underlying concept of heart disease on that particular data sample. This helps the FOLD algorithm to always pick up the relevant features regardless of the number of examples left.

The success of this approach highly depends on the choice of statistical algorithm as well as tuning its parameters to make sure that the model makes the fewest errors in its predictions. In this research we conducted all experiments using the "Extreme Gradient Boosting" (XGBoost) algorithm (Chen and Guestrin, 2016). XGBoost is an implementation of the Gradient Boosted Decision Tree algorithm. Although LIME is model agnostic, in the experiments presented in this chapter, XGBoost happened to always lead to better results.

Algorithm 7 shows how a standard tabular dataset is transformed into an ILP problem instance for the FOLD algorithm. This algorithm takes a dataset DS, a target predicate t, and a classifier model M that takes a feature vector and returns a binary classification value from the set {'+','-'}. For all data rows r in DS, there is an identifier that is denoted by $r.id$. The numeric features once discretized are sorted based on the produced intervals and the interval index in the sorted list is used as the second argument of such features in generating the background knowledge. For instance let a numeric feature such as blood pressure be discretized first and stored as a sorted list of intervals as follows:

$\{(-\infty, 97), [97, 120), [120, 153), [153, 170), [170, +\infty)\}$. The corresponding predicate for the datarow r with $r.id = 135$ and blood pressure value 130 is `blood_pressure(135,2)` because $135 \in [120, 153)$ whose index in the above list is 2.

Algorithm 7 Dataset Transformation with LIME

Input: t : target predicate,DS : Dataset
Input: M : trained classifier
Output: BK : background knowledge
Output: E^+,E^- : positive and negative examples

```
propositionalize categorical features
discretize numeric features
for each DataRow r ∈ DS do
    if M(r) ='+' then
        E+ = E+ ∪ {t(r.id)}
    else
        E− = E− ∪ {t(r.id)}
    end if
    explanation = LIME(M,r)
    for each pair(e, w) ∈ explanation do
        if e is the n^th discretized interval feature f then
            BK = BK ∪ {f(r.id, n)}
        end if
        if e is an equality expr. of the form f_v = 0 then
            // '-' denotes classical negation
            BK = BK ∪ {-f(r.id, v)}
        end if
        if  e is an equality expr. of the form f_v = 1 then
            BK = BK ∪ {f(r.id, v)}
        end if
    end for
end for
```

In Algorithm 7, explanation pairs with negative weights are retrieved too. These are the features that would turn the classification decision into the opposite of concept we are learning. For instance in Example 5.1, a healthy subject may happen to have a high level "serum cholesterol". Therefore, if LIME-FOLD algorithm picks up this feature to cover some positive examples, the healthy subjects—which are negative examples—are also covered.

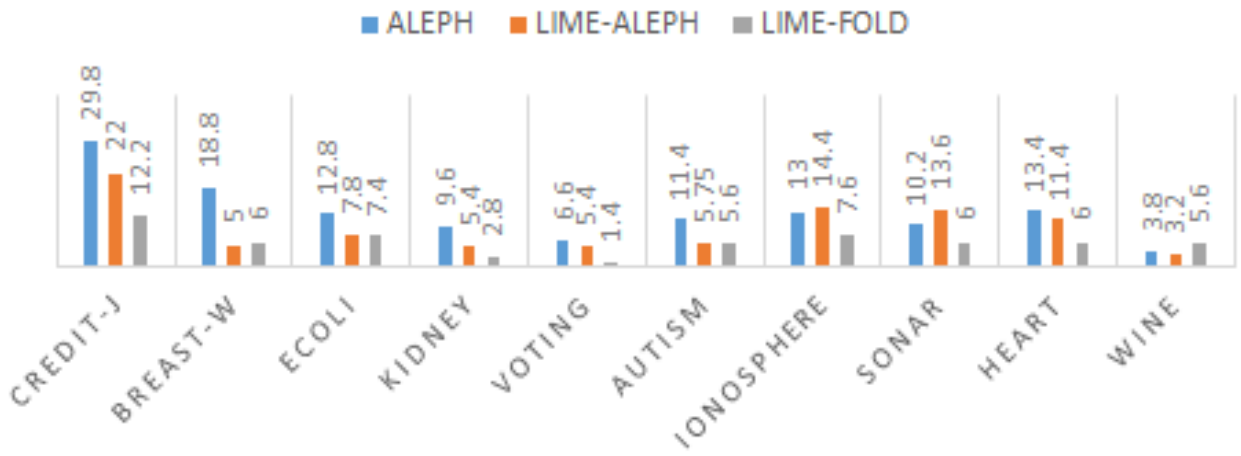


Figure 5.2: Average Number of Rules Induced by Each Different Experiment

The LIME-FOLD algorithm is able to rule these negative examples out by introducing an abnormality predicate that would make use of these negative weighted features. These are the features that led the XGBoost model to predict those subjects as healthy.

5.4 Experiments

In this section, we present our experiments on UCI standard benchmarks (Lichman, 2013b). The ALEPH system (Srinivasan, 2001) is used as the baseline. ALEPH is a state-of-the-art ILP system that has been widely used in prior work. To find a rule, ALEPH starts by building the most specific clause, which is called the "bottom clause", that entails a seed example. Then, it uses a branch-and-bound algorithm to perform a general-to-specific heuristic search for a subset of literals from the bottom clause to form a more general rule. We set ALEPH to use the heuristic enumeration strategy, and the maximum number of branch nodes to be explored in a branch-and-bound search to 500K. We use the standard metrics including precision, recall, accuracy and F_1 score to measure the quality of the results. We separately report the running time comparison as well. We conduct three different sets of experiments as follows: First, we run ALEPH on 10 different datasets using 5-fold cross-validation setting. Second, each dataset is transformed as explained in Algorithm 7. Then

the LIME-FOLD algorithm is run on a 5-fold cross-validated setting, and the classification metrics are reported. Third, ALEPH is run on the same datasets produced in the second experiment. We call this approach LIME-ALEPH.

Figure 5.2 compares the average number of clauses generated by standard ALEPH, LIME-ALEPH and LIME-FOLD on 10 UCI datasets. With the exception of "breast-w" and "wine", in all other datasets, LIME-FOLD discovers fewer number of clauses. However, in "breast-w" and "wine" the F_1 score of LIME-FOLD is higher than two other approaches. Also, it is worth noting that LIME-ALEPH in most cases generates fewer clauses than ALEPH. However, incorporating *Negation-As-Failure* in LIME-FOLD algorithm as well as learning the clauses in terms of defaults and exceptions allows the algorithm to cover all positive examples with fewer number of clauses.

Another observation that explains the advantage of LIME-ALEPH over ALEPH, is that LIME is capable of explaining propositionalized categorical variables in both affirmative and negative ways. For instance, in the "UCI heart" dataset, the *thallium-201 stress scintigraphy test* is a categorical feature with three possible values in the set {3,6,7}, indicating normal, fixed defect and reversible defect in that order. The covering approach incorporated in ALEPH, would come up with two clauses corresponding to 6, 7, whereas, in both LIME-ALEPH and LIME-FOLD a negated feature $f \neq 3$ is introduced and stored in the transformed dataset.

The following logic program is induced by LIME-FOLD algorithm (using the entire data set):

```
(1) heart_disease(A):- chest_pain(A,4), -thal(A,3).
(2) heart_disease(A):- slope(A,2), major_vessels(A,1).
(3) heart_disease(A):- chest_pain(A,4), sex(A,1),
                       not ab0(A).
(4) heart_disease(A):- blood_pressure(A,5), sex(A,1).
```

```
(5) heart_disease(A):- slope(A,2), blood_pressure(A,5).
(6) heart_disease(A):- slope(A,2), major_vessels(A,3),
                       serum_cholestoral(A,3).
    ab0(A):-major_vessels(A,3).
```

The induced program can be understood as follows: In clause (1), `chest_pain(A,4)` indicates an asymptomatic type of chest pain. While `thal(A,3)` would indicate a *thallium test* with normal results, the classically negated predicate `-thal(A,3)` indicates a proof that the *thallium test* is abnormal. In clause (2) `slope(A,2)` indicates *the slope of the peak exercise relative to rest* is flat, which is an indication of heart disease. `major_vessels(A,N)` indicates a patient with N (range: 0-3) colored major vessels during a Fluoroscopy test. The higher the number, the less narrowed major vessels. Clause (3) introduces an abnormality predicate which stipulates that an asymptomatic chest pain is an indication of heart disease unless there are no narrowed major vessels. High cholestrol and High blood pressure are specified in discretized intervals represented by their index. For instance `serum_cholestoral(A,3)` denotes the cholesterol range between 245 mg/dl and 400 mg/dl in this dataset. Similarly, `blood_pressure(A,5)` indicates systolic range between 15.7 and 18.6.

Figure 5.3 shows the "feature importance" plot calculated by xgboost algorithm. Generally, importance provides a score that indicates how useful or valuable each feature was in the construction of the boosted decision trees within the model. The more an attribute is used to make key decisions with decision trees, the higher its relative importance. Importance is calculated for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for. The LIME-FOLD approach, prefers the more "important" features over less "important" ones, because the weighted information gain heuristic scores clauses with more frequently used features higher. ALEPH induces 18 clauses on the same data. Some of the

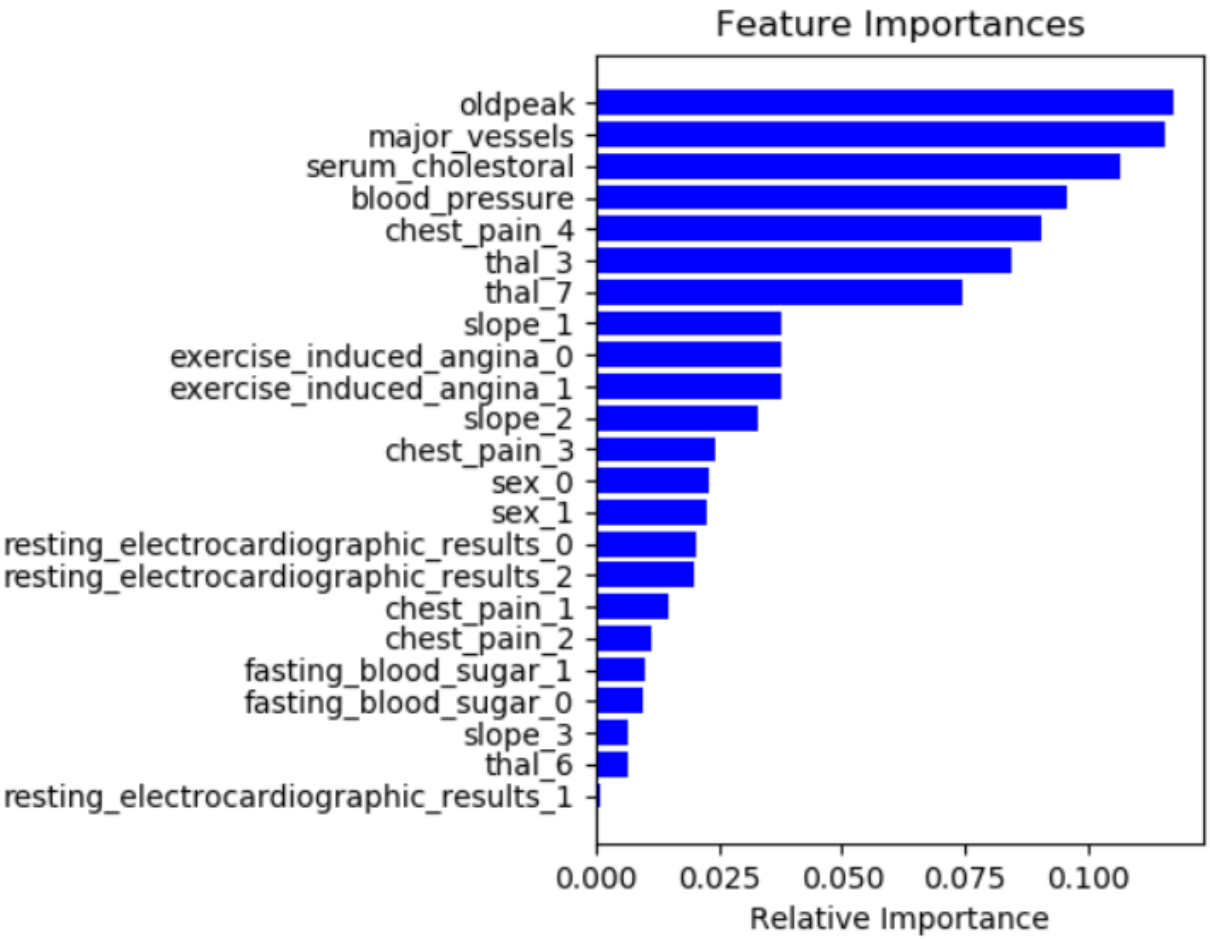


Figure 5.3: XGboost Feature Importance Plot for UCI Heart

Table 5.1: Evaluation of Our Three Experiments with 10 UCI Datasets

	Algorithm											
Data Set	Aleph				Aleph+Lime				**Fold+Lime**			
	Prec.	Recall	Acc.	F1	Prec.	Recall	Acc.	F1	Prec.	Recall	Acc.	F1
credit-j	0.78	0.72	0.78	0.75	**0.89**	0.69	0.82	0.77	0.86	**0.90**	**0.89**	**0.88**
breast-w	0.92	0.87	0.93	0.89	**0.98**	0.65	0.87	0.76	0.94	**0.92**	**0.95**	**0.92**
ecoli	0.85	0.75	0.84	0.80	0.95	0.84	0.92	0.89	**0.95**	**0.88**	**0.93**	**0.91**
kidney	0.96	0.92	0.93	0.94	**0.99**	0.95	**0.96**	**0.97**	0.93	**0.95**	0.93	0.94
voting	0.97	0.94	0.95	0.95	0.98	0.95	0.96	0.96	**0.98**	**0.96**	**0.97**	**0.97**
autism	0.73	0.43	0.79	0.53	**0.88**	0.38	0.81	0.52	0.84	**0.88**	**0.91**	**0.86**
ionosphere	0.89	0.87	0.85	0.88	0.92	0.85	0.86	0.88	**0.91**	**0.86**	**0.86**	**0.89**
sonar	0.74	0.56	0.66	0.64	0.81	0.72	0.74	0.76	**0.87**	**0.75**	**0.78**	**0.80**
heart	0.76	0.75	0.78	0.75	0.79	0.70	0.79	0.74	**0.82**	0.74	**0.82**	**0.78**
wine	0.94	**0.86**	0.93	0.89	0.91	0.85	0.92	0.88	**0.98**	0.85	**0.93**	**0.91**
Average	0.86	0.79	0.85	0.82	0.9	0.77	0.87	0.82	**0.92**	**0.87**	**0.91**	**0.89**

Table 5.2: Average Running Time Comparison

		Running Time (s)	
Data Set	size	ALEPH	LIME-FOLD
credit-j	125	1680	15
breast-w	699	83	7.8
ecoli	336	132	3
kidney	400	24	0.6
voting	435	252	1.8
autism	704	480	10.8
ionosphere	351	1080	4.8
sonar	208	834	9.6
heart	270	277	18.6
wine	178	18	1.8

features that the plot reports as rarely used by xgboost to split a node are introduced by ALEPH which makes the theory less relevant compared to what LIME-FOLD induces.

Table 5.2 compares the average running time of ALEPH against LIME-FOLD. For all 10 datasets, FOLD algorithm terminates in less than one minute. All experiments were run on an Intel Core i7 CPU @ 2.7GHz with 16 GB RAM and a 64-bit Windows 10. The FOLD algorithm is a Java application that uses JPL library to connect to SWI prolog. ALEPH v.5 has been ported into SWI-Prolog by (Riguzzi, 2016).

Table 5.1 presents the comparison of classification metrics on each of the 10 UCI datasets. The best performer is highlighted with boldface font. In 9 cases, the LIME-FOLD produces a classifier with higher F_1 score. However, in case of "kidney", LIME-ALEPH produces the highest F_1 score although, it generates almost twice as many clauses as LIME-FOLD does in this dataset.

5.5 Related Work

A survey of ILP can be found in (Muggleton et al., 2012). Rule extraction from statistical Machine Learning models has been a long-standing goal of the community. The rule extraction algorithms from machine learning models are classified into two categories: 1) Pedagogical

(i.e., learning symbolic rules from black-box classifiers without opening them) 2) Decompositional (i.e., to open the classifier and look into the internals). TREPAN (Craven and Shavlik, 1995) is a successful pedagogical algorithm that learns decision trees from neural networks. SVM+Prototypes (Núñez et al., 2002) is a decompositional rule extraction algorithm that makes use of KMeans clustering to extract rules from SVM classifiers by focusing on support vectors. Another rule extraction technique that is gaining attention recently is "RuleFit" (Friedman et al., 2008). RuleFit learns a set of weighted rules from ensemble of shallow decision trees combined with original features. In ILP community also, researchers have tried to combine statistical methods with ILP techniques. Support Vector ILP (Muggleton et al., 2005) uses ILP hypotheses as kernel in dual form of the SVM algorithm. kFOIL (Landwehr et al., 2006) learns an incremental kernel for SVM algorithm using a FOIL style specialization. nFOIL (Landwehr et al., 2005) integrates the Naive-Bayes algorithm with FOIL. The advantage of our research over all of the above mentioned research work is that, first it is model agnostic, second it is scalable thanks to the greedy nature of our clause search.

CHAPTER 6

WHITE-BOX INDUCTION FROM SUPPORT VECTOR MACHINES

6.1 Overview

The ILP learning problem can be regarded as a search problem for a set of clauses that deduce the training examples. The search is performed either top down or bottom-up. A bottom-up approach builds most-specific clauses from the training examples and searches the hypothesis space by using generalization. This approach is not applicable to large-scale datasets, nor it can incorporate *negation-as-failure* (Baral, 2003) into the hypotheses. A survey of bottom-up ILP systems and their shortcomings can be found at (Sakama, 2005). In contrast, top-down approach starts with the most general clauses and then specializes them. A top-down algorithm guided by heuristics is better suited for large-scale and/or noisy datasets (Zeng et al., 2014).

The FOIL algorithm by Quinlan (Quinlan, 1990b) is a popular top-down algorithm. FOIL uses heuristics from information theory called *weighted information gain*. The use of a greedy heuristic allows FOIL to run much faster than bottom-up approaches and scale up much better. However, scalability comes at the expense of losing accuracy if the algorithm is stuck in a local optima and/or when the number of examples is insufficient. The former is an inherent problem in hill climbing search and the latter is due to the shrinking of examples during clause specialization. Figure 6.1 demonstrates how the local optima results in discovering sub-optimal rules that do necessarily coincide with the real sub-concepts they are supposed to capture.

Unlike top-down ILP algorithms, Support Vector Machine (SVM) (Cortes and Vapnik, 1995) is a globally optimal learning method that generalizes very well and comes with test error upper-bound in terms of the number of support vectors and size of training input. However, this unique property is overshadowed by the black-box nature of SVM models.

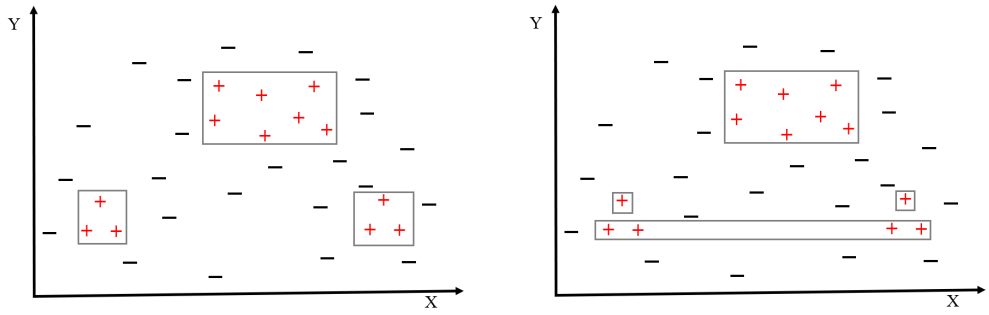


Figure 6.1: Optimal sequential covering with 3 Clauses (Left), Sub-Optimal sequential covering with 4 Clauses (Right)

Explaining the behavior of black-box models has motivated a long line of research in Rule Induction from SVM models. As we argue in more detail in section 4.5, all proposed Rule Extraction techniques either treat the model as black-box (Huysmans et al., 2008, 2006), or are limited to certain type of kernels (Fung et al., 2005), or are too complex to interpret (Nuñez et al., 2002). A survey of existing Rule Extraction techniques can be found in (Diederich, 2008).

Our new approach is based on the idea that each data sample is measurably similar/dissimilar to each support vector. Therefore, each support vector represents a subset of data samples. Now, if a set of features discriminate a support vector well, they would discriminate data samples similar to that support vector too. In order to measure the similarity and to pick up the support vector that is most similar to each data sample, we define a quantity based on the kernel value and each support vector's α parameter. To discover the most relevant features, our algorithm incorporates the SHAP technique. SHAP (Lundberg and Lee, 2017) is an example specific model interpreter that takes a model and an individual example and returns the contribution of each feature value in model's classification decision.

6.2 Support Vector Machines

Given a training dataset of m data points $\vec{x}_i \in \mathbb{R}^n$ and m corresponding labels $y_i \in \{1, -1\}$, the linear support vector machine is defined as the following optimization problem:

$$\min_{\vec{w},b,\xi_i \geq 0} \frac{1}{2}\|\vec{w}\|_2^2 + C\sum_{i=1}^{m}\xi_i \tag{6.1}$$

such that: $y_i(\vec{w}.\vec{x}_i - b) + \xi_i \geq 1$ where ξ_i is the slack error to potentially allow some points to be misclassified, $\vec{w}$ is the perpendicular vector to the separating hyper-plane, b is the offset of that hyper-plane, C is a hyperparameter and determines the degree to which misclassification is allowed to avoid over-fitting.

An equivalent yet more efficient form of SVM problem known as dual formulation is defined as follows:

$$\max_{\alpha_i} -\frac{1}{2}\sum_{i=1}^{m} y_i y_j \vec{x}_i \vec{x}_j \alpha_i \alpha_j + \sum_{i=1}^{m}\alpha_i \tag{6.2}$$

such that $\sum_{i=1}^{m}\alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$

The dot product in dual formulation can be replaced with any kernel function (i.e., a function that maps data into higher dimensions). For non-linearly separable data, Kernels map the data into a higher dimensional feature space where data becomes separable. Equation 6.2 is a special case for the following general dual formulation:

$$\max_{\alpha_i} -\frac{1}{2}\sum_{i=1}^{m} y_i y_j K(\vec{x}_i, \vec{x}_j) \alpha_i \alpha_j + \sum_{i=1}^{m}\alpha_i \tag{6.3}$$

such that $\sum_{i=1}^{m}\alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$

After solving the above problem using quadratic programming, the α_i and b are used to classify a new data sample $\vec{x}$ as follows:

$$f(x) = sign\left[\sum_{i=1}^{m}\alpha_i y_i K(\vec{x}_i, \vec{x}) + b\right] \tag{6.4}$$

It turns out that α is a sparse vector and only few α_i come back with non-zero values from the quadratic solver package. They are called the support vectors and as Equation 6.4 suggests, proportionate to their respective α_i value, support vectors are the only influential data points in classification decision of any new data sample.

While the kernel function is meant to map data points into a higher dimension efficiently, one can also interpret the kernel value $K(\vec{x}_i, \vec{x}_j)$ as a similarity measure between points $\vec{x}_i$ and $\vec{x}_j$. For instance, in case of the Gaussian radial basis kernel (rbf):

$$K(\vec{x}_i, \vec{x}_j) = e^{-\gamma \|\vec{x}_i - \vec{x}_j\|_2^2} \tag{6.5}$$

where γ is a hyper parameter. If $\vec{x}_i$ and $\vec{x}_j$ are similar, the kernel value would be close to 1. Otherwise, it would be close to 0. This can be naturally used to quantify similarity. However, it should be noted from Equation 6.4 that the magnitude of α_i also contributes to the influence and similarity. Therefore, we define the similarity of data point $\vec{x}$ and the i^{th} support vector $\vec{x}_i$ as follows:

$$sim_i(\vec{x}) = \alpha_i y_i K(\vec{x}_i, \vec{x}) \tag{6.6}$$

For any new data sample $\vec{x}$ the support vector with highest sim value is the one that contributes most to the prediction of $\vec{x}$. Figure 6.2 demonstrates an SVM model with rbf kernel trained on the same dataset from Figure 6.1. In this figure, support vectors are the dots located on the dashed lines. They are labeled with an integer identifier. Every other data point is annotated with the identifier of the most similar support vector calculated using Equation 6.6. Since the similarity is measured with respect to the concept being learned, the most similar support vector does not necessarily accord with the Euclidean distance.

6.3 SHAP

SHAP (Lundberg and Lee, 2017) (SHapley Additive exPlanations) is a unified approach with foundations in game theory to explain the output of any machine learning model in terms of its features' contributions. To compute each feature i's contribution, SHAP requires retraining the model on all feature subsets $S \subseteq F$, where F is the set of all features. For any feature i, a model $f_{S\cup\{i\}}$ is trained with the feature i present, and another model f_S is trained with feature i eliminated. Then, the difference between predictions is computed as follows:

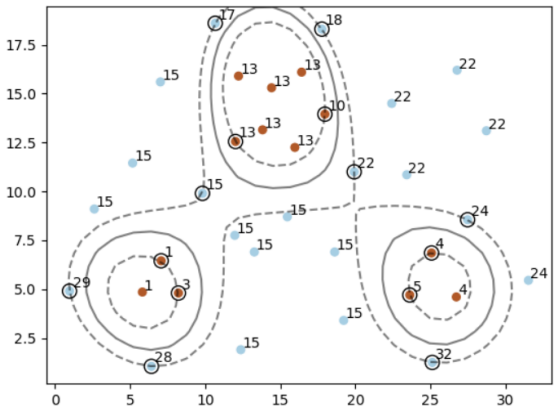


Figure 6.2: Annotating Data Points in a 2D dataset With Most Similar Support Vector

$f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)$, where x_S represents sample's feature values in S. Since the effect of withholding a feature depends on other features in the model, the above differences are computed for all possible subsets of $S \subseteq F \setminus \{i\}$ and their average taken. The weighted average of all possible differences (a.k.a Shapley value) is used as feature importance. Equation 6.7 shows how Shapley value associated with each feature value is computed:

$$\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \Big[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S) \Big] \tag{6.7}$$

Given a dataset and a trained model, SHAP outputs a matrix with the shape $(\#samples, \#features)$ representing the Shapley value of each feature for each data sample. Each row sums to the difference between the model output for that sample and the expected value of the model output. This difference explains why the model is inclined on predicting a specific class outcome.

Example 6.1. *The UCI heart dataset contains features such as patient's blood pressure, chest pain, thallium test results, number of major vessels blocked, etc. The classification task is to predict whether the subject suffers from heart disease or not. Figure 6.3 shows how SHAP would explain a model's prediction over a data sample.*

For this individual, SHAP explains why the model predicts heart disease by returning the top features along with their Shapley values (importance weight). According to SHAP, the

model predicts "heart disease" because of the values of "thalium test" and "maximum heart rate achieved" which push the prediction from the base (expected) value of 0.44 towards a positive prediction (heart disease). On the other hand, the feature "chest pain" would have pushed the prediction towards negative (healthy), but it is not strong enough to turn the prediction.

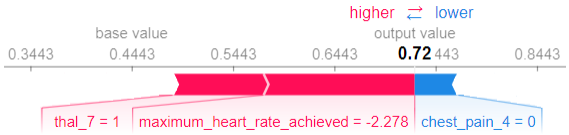


Figure 6.3: Shap Values for A UCI Heart Prediction

The categorical features should be *binarized* before an SVM model can be trained. Binarization (aka one-hot encoding) is the process of transforming each categorical feature with domain of cardinality n, into n new binary predicates (features).In Example 6.1, chest pain level is a categorical feature with 4 different values in the set $\{1, 2, 3, 4\}$. Type 4 chest pain indicates asymptomatic pain and is a serious indication of a heart condition. In this case, binarization results in 4 different predicates. The "thalium test" is also a categorical feature with outcomes in the set {3,6,7}. Any outcome other than 3, indicates a defect (6 for fixed and reversible for 7).

In case of Example 6.1, SHAP determines that the feature "thal_7" with outcome of 1 (True), pushes the prediction towards heart disease. To reflect this fact in our ILP algorithm, for any person X, the predicate `thal(X,7)` is introduced. Also, SHAP indicates that the binary feature "chest_pain_4" with value 0 (False), pushes the prediction towards healthy. In our ILP algorithm this is represented by *negation-as-failure* (Baral, 2003) as `not chest_pain(X,4)`.

6.4 SHAP-FOIL

In this section we introduce SHAP_FOIL, an algorithm capable of learning non-monotonic logic programs based on the global behavior of an SVM model.

There are two major issues with the *sequential covering* algorithms such as FOIL: 1) As number of examples decreases during specialization loop, probability of introducing an irrelevant predicate that accidentally splits a particular set of examples increases. 2) The greedy nature of *hill-climbing* search in clause specialization, sometimes results in introduction of wrong predicates that would cover more examples at a certain moment, but eventually leads to inducing a clause that does not perfectly represent sub-concepts as shown in Example 6.1. This is known as Local Optima problem in *hill-climbing* search. Determining the sub-concepts requires a global view which could only happen via a global optimization process such as an SVM model. However, finding the best separating hyperplane in a higher dimension does not explain the contributing features in any classification decision made by the model.

SHAP is able to quantitatively explain the features that would push the model towards predicting a specific outcome. In particular, for each support vector, SHAP determines a subset of feature value pairs that would make the model arrive at a certain decision. It turns out that just by having the Shapley values of support vectors, our algorithm can learn the global underlying behavior of SVM model. This is mathematically justified as follows: From Equation 6.4, every new data sample is interpreted in terms of similarity to support vectors. The internal points are not relevant (because their corresponding α_i parameter is 0). Among support vectors, only the ones that are closely "similar" to the given point are relevant (because, for dissimilar support vectors the kernel value of Equation 6.5 is close to 0).

The intuition behind Shap-FOIL algorithm is as follows: If a subset of feature-values explains the decision on a particular support vector, it explains the decision on data points

that are "similar" to that support vector too. Similarity is measured using Equation 6.6. In the context of *sequential covering* scheme, SHAP_FOIL would find the support vector that pulls the greatest number of data points in terms of Equation 6.6. Then, it would specialize a clause by introducing predicates that are determined by SHAP for that support vector. Then, the algorithm removes the data points that are covered by that rule. It also removes the support vector. This process is repeated for the remaining data points. Since there are only finite number of support vectors, the algorithm is guaranteed to terminate.

Unlike most ILP algorithms, SHAP_FOIL does not require discretization of numerical features in advance. During the specialization of a clause, if a numeric feature happens to have the highest Shapley number for a support vector, a real arithmetic constraint is introduced by the algorithm. The end-points of this interval is determined by looking into the respective values of all data points that are most similar to that support vector.

Algorithm 8 summarizes the SHAP_FOIL algorithm. The algorithm inputs are as follows: 1) Set D of cardinality m representing m training data points. 2) SHAP matrix of a trained SVM model on D. 3) A threshold Θ to determine minimum acceptable accuracy of each induced clause. 4) Set SV that are True Positive (TP) support vectors (i.e., support vectors with label 1, also predicted 1 by the SVM model). The model outputs a hypothesis comprising a set of induced clauses. The While loop in line 2, iterates until all support vectors are considered. (termination condition). In line 3, by calling the function ANNOTATE_SAMPLES, all remaining datapoints are annotated with a support vector among the remaining support vectors. This support vector must have the highest similarity value to that data point. In line 4, the algorithm chooses sv, the support vector which pulled the greatest number of data points from annotation. This allows to discover the more inclusive rules first. In lines 5 and 6, similar to the FOIL algorithm, specialization from the most general clause (i.e., `target :- true.`) is conducted. To specialize a clause, predicates determined by the Shapley value of sv are added to the body in the order of their Shapley

Algorithm 8 Summarizing the SHAP_FOIL algorithm

Input: $D = \{(\vec{x}_1, y_1), ..., (\vec{x}_m, y_m)\}$
Input: $SHAP\ matrix\ (m, \#features), \Theta$
Input: $S = \{sv \mid sv\ is\ a\ support\ vector\ and\ sv\ is\ TP\}$
Output: $Hypothesis\ H = \{\}$

```
1: function SHAP_FOIL(S, D)
2:     while (S ≠ ∅) do
3:         sim_map = ANNOTATE_SAMPLES(S,D)
4:         sv = argmax_{s∈S} len(sim_map[s])
5:         c ← (target :- true.)
6:         ĉ ← Specialize c using SHAP[sv]
7:         if ĉ's accuracy on D ≥ Θ then
8:             H ← H ∪ {ĉ}
9:             D ← D − {x⃗_i | x⃗_i ∈ D ∧ ĉ ⊨ y_i}
10:        end if
11:        S ← S − {sv}
12:    end while
13:    return H
14: end function
15: function ANNOTATE_SAMPLES(S, D)
16:    Let sim_map be a map of type : S ↦ List
17:    for each sv ∈ S do
18:        sim_map[sv] = [ ]
19:    end for
20:    for each x⃗_i ∈ D do
21:        sv = argmax_{s∈S} sim(s, x⃗_i)
22:        sim_map[sv].append(x⃗_i)
23:    end for
24:    return sim_map
25: end function
```

value magnitude. To add numeric features, our algorithm creates an interval by finding the smallest and largest values in the list of data samples associated with sv. The specialized clause is named $\hat{c}$. In line 7, the accuracy of $\hat{c}$ is tested against a threshold Θ. If it is higher than Θ, $\hat{c}$ is added to the current hypothesis H in line 8. In line 9, similar to FOIL, the set of data points covered by $\hat{c}$ are removed from D (*sequential covering*). If $\hat{c}$ achieves lower accuracy than Θ, it is discarded. Regardless of the case, in line 11, sv is removed from the set of support vectors. This serves two purposes: 1) It guarantees the termination. 2) More

importantly, if in some iteration, a support vector pulls greatest number of similar points but it does not yield an above Θ accurate clause, to make progress possible, this support vector will be removed from consideration. We will clarify this more in Example 6.3.

Example 6.2. *Figure 6.1 illustrates the local optima issue of FOIL. In Figure 6.2, an SVM model is shown for the same dataset with two features f1 and f2 and two classes of red and blue.The SHAP_FOIL algorithm learns the following logic program on this dataset:*

```
red(X):- f1(X,F1), 12.02 =< F1 <= 17.97,
         f2(X,F2), 12.25 =< F2 <= 16.1 .
red(X):- f1(X,F1), 5.82 =< F1 <= 8.22,
         f2(X,F2), 4.8 =< F2 <= 6.45 .
red(X):- f1(X,F1), 23.62 =< F1 <= 26.72,
         f2(X,F2), 4.6 =< F2 <= 6.85 .
```

Example 6.3. *To add numeric features, our algorithm creates an interval by finding the smallest and largest values in the list of data samples associated with a support vector sv. This approach sometimes results in too coarse-grained intervals that cover too many False Positives (FP) to tolerate. As explained earlier, to handle this case, SHAP_FOIL, removes sv and tries to break the region into smaller sub-regions. Each smaller region is then covered by other support vectors. Figure 6.4, illustrates this with a dataset of two features on which an SVM model is trained. At iteration 1, after annotating the data samples, the support vector 3, pulls the majority of data samples. According to SHAP, both numeric features contribute to the classification decision. However, after creating the intervals for both features, the clause shown as a green box, ends-up covering significant number of False Positives. This is shown in Figure 6.4. Therefore, the clause is discarded and the support vector 3, is removed from consideration. As shown in Figure 6.5, on iteration 2, the support vector 8 pulls the highest number of data samples. It yields an accurate clause. Thus, it is added to the hypothesis.*

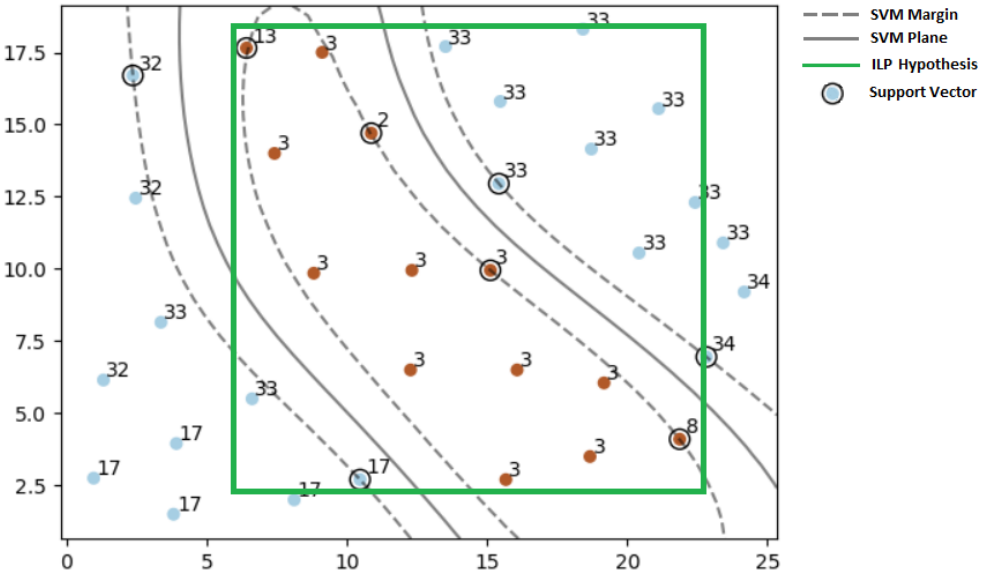


Figure 6.4: Iteration #1 of Example 6.3

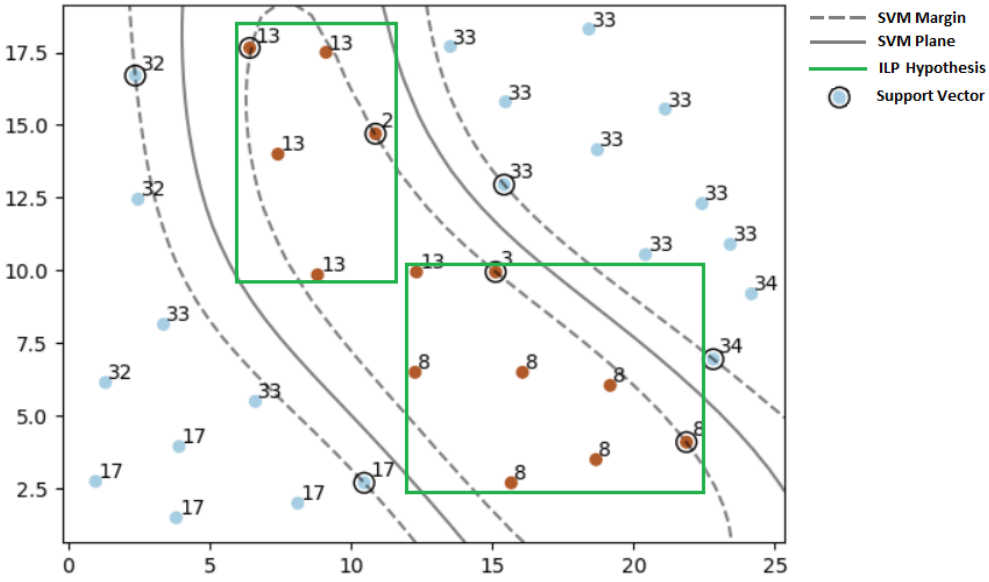


Figure 6.5: Iteration #2, #3 of Example 6.3

On iteration 3, the support vector 2 pulls the rest of data samples and once again, it results in an accurate clause.

Table 6.1: Evaluation of SHAP_FOIL on UCI Datasets

		Algorithm											
		SVM				Aleph				SHAP-FOIL			
Data Set	Kernel	Prec.	Recall	Acc.	F1	Prec.	Recall	Acc.	F1	Prec.	Recall	Acc.	F1
credit-j	rbf	0.84	0.84	0.84	0.84	0.78	0.72	0.78	0.75	**0.83**	**0.76**	**0.83**	**0.80**
breast-w	Poly	0.97	0.96	0.96	0.96	0.92	0.87	0.93	0.89	**0.97**	**0.89**	**0.95**	**0.93**
ecoli	rbf	0.96	0.96	0.96	0.96	0.85	0.75	0.84	0.80	**0.86**	**0.94**	**0.89**	**0.90**
kidney	poly	0.99	0.99	0.99	0.99	0.96	0.92	0.93	0.94	**0.97**	**0.97**	**0.97**	**0.97**
voting	rbf	0.95	0.94	0.94	0.94	**0.97**	**0.94**	**0.95**	**0.95**	0.92	0.94	0.91	0.93
autism	rbf	1.00	1.00	1.00	1.00	0.73	0.43	0.79	0.53	**0.94**	**0.86**	**0.94**	**0.88**
ionosphere	rbf	0.95	0.95	0.94	0.95	0.89	0.87	0.85	0.88	**0.92**	**0.90**	**0.90**	**0.91**
heart	poly	0.81	0.80	0.80	0.80	0.76	0.75	0.78	0.75	**0.90**	**0.86**	**0.90**	**0.88**

6.5 Experiments

In this section, we present our experiments on UCI standard benchmarks (Lichman, 2013b). The ALEPH system (Srinivasan, 2001) is used as the baseline. ALEPH is a state-of-the-art ILP system that has been widely used in prior work. To find a rule, ALEPH starts by building the most specific clause, which is called the "bottom clause", that entails a seed example. Then, it uses a branch-and-bound algorithm to perform a general-to-specific heuristic search for a subset of literals from the bottom clause to form a more general rule. We set ALEPH to use the heuristic enumeration strategy, and the maximum number of branch nodes to be explored in a branch-and-bound search to 500K. We use the standard metrics including precision, recall, accuracy and F_1 score to measure the quality of the results.

The *sequential-covering* based algorithms - including ALEPH and FOIL - tend to learn too many rules in presence of noisy data. Both algorithms induce more accurate clauses at the expense of covering fewer examples by each clause. In our SHAP_FOIL algorithm, specialization is stopped once the purity of a clause reaches the threshold Θ while the maximum coverage is guaranteed by SHAP because the specialization is performed in the order of features Shapley number. For instance, While ALEPH discovers 15 clauses for UCI heart, the following logic program comprised of only 6 clauses is induced by the SHAP_FOIL algorithm:

```
(1) heart_disease(X) :-
            thallium_test(X,7),
```

```
            chest_pain(X,4),
            exercise_induced_angina(X).

(2) heart_disease(X) :-
            maximum_heart_rate_achieved(X,F1),
            106 =< F1, F1 =< 154,
            not major_vessels(X,0),
            oldpeak(X,F2),
            1 =< F2, F2 =< 4.
(3) heart_disease(X) :-
            not major_vessels(X,0),
            thallium_test(X,7),
            chest_pain(X,4).
(4) heart_disease(X) :-
            thallium_test(X,7),
            age(X,F1),
            35 =< F1, F1 =< 52,
            chest_pain(X,4).
(5) heart_disease(X) :-
            maximum_heart_rate_achieved(X,F1),
            120 =< F1, F1 =< 147,
            exercise_induced_angina(X),
            chest_pain(X,4).
(6) heart_disease(X) :-
            not major_vessels(X,0),
            chest_pain(X,4),
```

```
male(X).
```

The induced program can be understood as follows: In clause (1), `thallium_test(X,7)` indicates a *thallium test* with reversible defect, while `chest_pain(X,4)` indicates an asymptomatic type of chest pain. According to clause (1), these two conditions, conjoined with angina revealed in an exercise test indicate the existence of heart disease. In clause (2), if maximum heart rate achieved and during exercise test falls in the discovered range 106-154 and ST depression induced by exercise relative to rest falls in the range 1-4 and there are signs of blockage in major vessels (indicated by *negation-as-failure*), the combination means heart disease. The rest of the clauses are read similarly.

Table 6.1 presents the comparison between ALEPH and SHAP_FOIL on classification evaluation of each UCI dataset. The best performer is highlighted with boldface font. With the exception of congressional voting dataset where the SVM performance is lower than ALEPH, the SHAP_FOIL algorithm always achieves higher score compared to ALEPH. Note that our SHAP_FOIL algorithm not only does better than ALEPH in classification evaluation measures, it also produces much smaller number of rules. In many cases, ALEPH produces an order of magnitude more rules than the SHAP_FOIL algorithm. Smaller number of rules are more readily understood by the user. They can be manually revised by the user much more easily as well (to better capture the learned knowledge) based on user's background knowledge about the problem.

6.6 Related Works

A survey of ILP can be found in (Muggleton et al., 2012). Rule extraction from statistical Machine Learning models has been a long-standing goal of the community. The rule extraction algorithms from machine learning models are classified into two categories: 1) Pedagogical (i.e., learning symbolic rules from black-box classifiers without opening them) 2) Decompositional (i.e., to open the classifier and look into the internals). TREPAN (Craven and

Shavlik, 1995) is a successful pedagogical algorithm that learns decision trees from neural networks. Minerva (Huysmans et al., 2008) and Iter (Huysmans et al., 2006) are pedagogical approaches to extract rules from SVM models. There is also a broader pedagogical approach to rule extraction where an SVM model is trained first, then the entire training data is re-labeled using the predictions of the SVM model, and finally a rule learning method (e.g., C4.5, ID3, CART etc.) is applied. It should be noted that this approach suffers from the very issue of local optima discussed in Figure 6.1.

SVM+Prototypes (Núñez et al., 2002) is a decompositional rule extraction algorithm that makes use of KMeans clustering to extract rules from SVM classifiers by focusing on support vectors. The main drawback of this approach is that all extracted rules contain all possible input variables in its conditions, making the approach too complex to interpret for large input dimensions. Fung (Fung et al., 2005) is another decompositional SVM rule extraction technique to extract propositional rules which is limited to linear kernels. The advantage of our SHAP_FOIL algorithm is that it can handle all kernels and the induced hypotheses are expressed in terms of the original features.

CHAPTER 7

INDUCTION OF LOGIC PROGRAMS FROM MACHINE LEARNING MODELS USING HIGH-UTILITY ITEM-SET MINING

7.1 Overview

The ILP learning problem can be regarded as a search problem for a set of clauses that deduce the training examples. The search is performed either top down or bottom-up. A bottom-up approach builds most-specific clauses from the training examples and searches the hypothesis space by using generalization. This approach is not applicable to large-scale datasets, nor it can incorporate *negation-as-failure* into the hypotheses. A survey of bottom-up ILP systems and their shortcomings can be found at (Sakama, 2005). In contrast, top-down approach starts with the most general clause and then specializes it. A top-down algorithm guided by heuristics is better suited for large-scale and/or noisy datasets (Zeng et al., 2014).

The FOIL algorithm by Quinlan (Quinlan, 1990b) is a popular top-down algorithm. FOIL uses heuristics from information theory called *weighted information gain*. The use of a greedy heuristic allows FOIL to run much faster than bottom-up approaches and scale up much better. However, scalability comes at the expense of losing accuracy if the algorithm is stuck in a local optima and/or when the number of examples is insufficient. Figure 7.1 demonstrates how the local optima results in discovering sub-optimal rules that does not necessarily coincide with the real underlying sub-concepts of the data. Additionally, since the objective is to learn pure clauses (i.e., clauses with zero or few negative example coverage) the FOIL algorithm often discovers too many clauses each of which only cover a few examples. Discovery of a huge number of clauses reduces the interpretability and also it does not generalize well on the test data.

Unlike top-down ILP algorithms, statistical machine learning algorithms are bound to find the relevant features because they optimize an objective function with respect to global

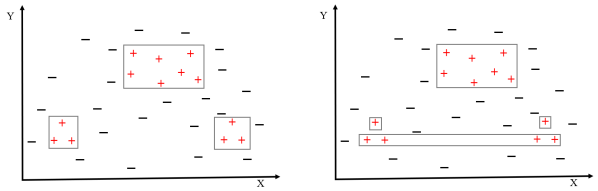


Figure 7.1: Optimal sequential covering with 3 Clauses (Left), Sub-Optimal sequential covering with 4 Clauses (Right)

constraints. This results in models that are inherently complex and cannot explain what features account for a classification decision on any given data sample. The Explainable AI techniques such as LIME (Ribeiro et al., 2016) and SHAP (Lundberg and Lee, 2017) have been proposed that provide explanations for any given data sample. Each explanation is a set of feature-value pairs that would locally determine what features and how strongly each feature, relative to other features, contributes to the classification decision. To capture the global behavior of a black-box model, however, an algorithm needs to group similar data samples (i.e., data samples for which the same set of feature values are responsible for the choice of classification) and cover them with the same clause. While in FOIL, the search for a clause is guided by heuristics, in our novel approach, we adapt *High Utility Item-set Mining* (HUIM) (Gan et al., 2018) — a popular technique from data mining — to find clauses. We call this algorithm SHAP-FOLD from here on. The advantage of SHAP-FOLD over heuristics-based algorithms such as FOIL is that:

1. SHAP-FOLD does not get stuck in a local optima

2. SHAP-FOLD distinguishes exceptional cases from noisy samples

3. SHAP-FOLD learns a reasonable number of non-monotonic rules in the form of default theories that would understandably capture the global behavior of any black-box model

4. SHAP-FOLD is fast and scalable compared to conventional ILP algorithms

This chapter makes the following novel contribution: We present a new ILP algorithm capable of learning *non-monotonic* logic programs from local explanations of black-box models provided by SHAP. Our experiments on UCI standard benchmark data sets suggest that SHAP-FOLD outperforms ALEPH (Srinivasan, 2001) in terms of classification evaluation metrics, running time, and providing more concise explanations measured in terms of number of clauses induced.

7.2 High-Utility Itemset Mining

The problem of *High-Utility Itemset Mining* (HUIM) is an extension of an older problem in data mining known as *frequent pattern mining* (Aggarwal and Han, 2014). Frequent pattern mining is meant to find frequent patterns in transaction databases. A *transaction database* is a set of records (transactions) indicating the items purchased by customers at different times. A *frequent itemset* is a group of items that appear in many transactions. For instance, {noodles, spicy sauce} being a frequent itemset, can be used to take marketing decisions such as co-promoting noodles with spicy sauce. Finding frequent itemsets is a well-studied problem with an efficient algorithm named Apriori (Agrawal and Srikant, 1994). However, in some applications frequency is not always the objective. For example, the pattern {milk,bread} may be highly frequent, but it may yield a low profit. On the other hand, a pattern such as {caviar, champagne} may not be frequent but may yield a high profit. Hence, to find interesting patterns in data, other aspects such as profitability is considered.

Mining high utility itemsets can be viewd as a generalization of the frequent itemset mining where each item in each transaction has a utility (importance) associated with it and the goal is to find itemsets that generate high profit when for instance, they are sold together. The user has to provide a value for a threshold called *minimum utility*. A high

utility itemset mining algorithm outputs all the high-utility itemsets with at least *minimum utility* profit. The HUIM problem is formally defined as follows:

- $I = \{i_1, i_2, ..., i_m\}$ is a set of items.
- $D = \{T_1, T_2, ..., T_n\}$ be a transaction database where each transaction $T_i \in D$ is a subset of I.
- $u(i_p, T_q)$ denotes the utility (profit) for item i_p in transaction T_q. For example $u(b, T_0) = 10$ in Example from Table 7.1.
- $u(X, T_q)$, utility of an itemset X is defined as $\sum_{i_p \in X} u(i_p, T_q)$, where $X = \{i_1, i_2, ..., i_K\}$ is a k-itemset, $X \subseteq T_q$ and $1 \leq K \leq m$.
- $u(X)$, utility of an itemset X is defined as $\sum_{T_q \in D \wedge X \subseteq T_q} u(X, T_q)$.

An itemset X is a *high utility itemset* if $u(X) \geq \varepsilon$, where $X \subseteq I$ and ε is the *minimum utility* threshold, otherwise, it is a low utility itemset. Table 7.1 shows a transaction database consisting of 5 transactions. Left columns shows the transaction Identifier. Middle column contains the items included in each transaction and right column contains each item's respective profit. If the ε is set to 25, the result of a high utility itemset mining algorithm is shown in the right table in Table 7.1.

Several high utility itemset mining algorithms have been proposed (e.g., UMining, Two-Phase, IHUP, UP-Growth, etc). A complete survey of these algorithms can be found in (Fournier-Viger et al., 2019). The differences between these algorithms lies in the data structures, strategies that are employed for searching high utility itemsets (DFS vs. BFS), and how they prune the unpromising paths.

The downside of this approach is that it requires the decision maker to choose a *minimum utility* threshold value for discovering interesting itemsets. This is quite challenging as too low a choice of ε results in too many itemsets and too high a choice of ε results in too

Table 7.1: Left: A High Utility Itemset Problem Instance. Right: Solution for minutil = 25

Transactions	Items	Profits
$\mathbf{T_0}$	a b c d e	5 10 1 6 3
$\mathbf{T_1}$	b c d e	8 3 6 3
$\mathbf{T_2}$	a c d	5 1 2
$\mathbf{T_3}$	a c e	10 6 6
$\mathbf{T_4}$	b c e	4 2 3

High Utility Itemsets	
{a, c}: 28	{a, c, e}: 31
{a, b, c, d, e}: 25	{b, c}: 28
{b, c, d}: 34	{b, c, d, e}: 40
{b, c, e}: 37	{b, d}: 30
{b, d, e}: 36	{b, e}: 31
{c, e}: 27	

few itemsets. In order to address this issue, Top-K High Utility Itemset (THUI) mining problem was introduced (Tseng et al., 2016), where the user wants to discover the k itemsets having the highest utility. A top-k high-utility itemset mining algorithm works as follows: It initially sets an internal ε threshold to 0, and starts to explore the search space. Then, as soon as k high utility itemsets are found, the internal ε is raised to the utility of the pattern having the lowest utility among the current top-k patterns. Then, the search continues and for each high utility itemset found, the set of the current top-k pattern is updated as well as the internal ε. When the algorithm terminates, it returns the set of the top-k high utility itemsets.

7.3 SHAP-FOLD Algorithm

In this section we present the SHAP-FOLD algorithm. SHAP-FOLD learns a concept in terms of a default theory (Shakerin et al., 2017). A default theory is a *non-monotonic* logic theory to formalize reasoning with default assumptions in absence of complete information. In Logic Programming, default theories are represented using *negation-as-failure* (NAF) semantics (Baral, 2003).

Example 7.1. *The following default theory "Normally, birds fly except penguins which do not", is represented as:*

```
flies(X) :- bird(X), not ab_bird(X).
ab_bird(X) :- penguin(X).
```

This default theory is read as: "For every object X, X flies if X is a bird and is not abnormal. For every object X, X is an abnormal bird if it is a penguin".

The SHAP-FOLD algorithm adapts the FOIL style sequential covering scheme. Therefore, it iteratively learns single clauses, until all positive examples are covered. To learn one clause, SHAP-FOLD first finds common patterns among positive examples. If the resulted clause (default) covers a significant number of negative examples, SHAP-FOLD swaps the current positive and negative examples and recursively calls the algorithm to learn common patterns in negative examples (exceptions). As shown in Example 7.1, the exceptions are ruled out using negation-as-failure. Learning exceptions allow our SHAP-FOLD algorithm to distinguish between noisy samples and exceptional cases.

To search for "best" clause, SHAP-FOLD tightly integrates the High Utility Itemset Mining (HUIM) and the SHAP technique. In this novel approach, the SHAP system is employed to find relevant features as well as their importance.

Example 7.2. *The UCI heart dataset contains features such as patient's blood pressure, chest pain, thallium test results, number of major vessels blocked, etc. The classification task is to predict whether the subject suffers from heart disease or not. Figure 7.2 shows how SHAP would explain a model's prediction over a data sample.*

For this individual, SHAP explains why the model predicts heart disease by returning the top features along with their Shapley values (importance weight). According to SHAP, the model predicts "heart disease" because of the values of "thalium test" and "maximum heart rate achieved" which push the prediction from the base (expected) value of 0.44 towards a positive prediction (heart disease). On the other hand, the feature "chest pain" would have pushed the prediction towards negative (healthy), but it is not strong enough to turn the prediction.

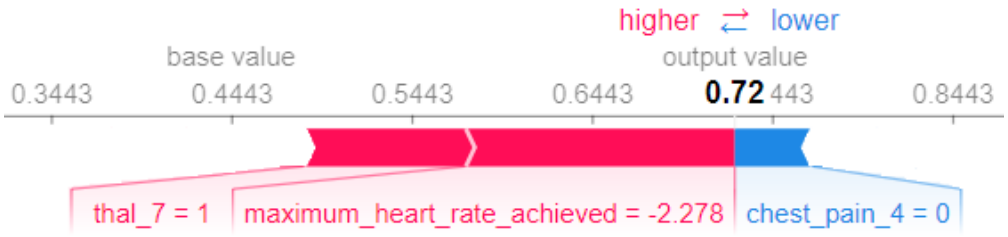


Figure 7.2: Shap Values for A UCI Heart Prediction

The categorical features should be *binarized* before any model is trained. Binarization (a.k.a one-hot encoding) is the process of transforming each categorical feature with domain of cardinality N, into N new binary predicates (features).In Example 7.2, chest pain level is a categorical feature with 4 different values in the set $\{1, 2, 3, 4\}$. Type 4 chest pain indicates asymptomatic pain and is a serious indication of a heart condition. In this case, binarization results in 4 different predicates. The "thalium test" is also a categorical feature with outcomes in the set {3,6,7}. Any outcome other than 3, indicates a defect (6 for fixed and reversible for 7). In case of Example 5.1, SHAP determines that the feature "thal_7" with outcome of 1 (True), pushes the prediction towards heart disease. To reflect this fact in our ILP algorithm, for any person X, the predicate `thal(X,7)` is introduced. Also, SHAP indicates that the binary feature "chest_pain_4" with value 0 (False), pushes the prediction towards healthy.

To find the "best" clause SHAP-FOLD creates instances of HUIM problem. Each instance, contains a subset of examples represented as a set of "transactions" as shown in Table 7.1. Each "transaction" contains a subset of feature values along with their corresponding utility (i.e., feature importance). The feature importance $\phi_i \in [0, 1]$ for all i distinct feature values. Therefore, a *high-utility itemset* in any set of "transactions" represents strongest features that would contribute to the classification of a significant number of examples, because, otherwise, that itemset would not have been selected as a high-utility itemset. To find the itemset with highest utility, the HUIM algorithm Top-K (Tseng et al., 2016) is invoked with K set to 1.

SHAP-FOLD takes a target predicate name (G), a tabular dataset (D) with m rows and two different labels $+1$ and -1 for positive examples and negative examples respectively. E^+ and E^- represent these examples in the form of target atoms. It also takes a "transaction" database. Each row of T contains a subset of an example's feature-values ($\vec{z}_i$) along with their Shapley values ($\vec{\phi}_i$). This "transaction" database is passed along to create HUIM instance and find the itemset with highest utility every time Top-K algorithm is invoked. The summary of SHAP-FOLD's pseudo-code is shown in Algorithm 9.

In the function FOIL (lines 1-8), *sequential covering* loop to cover positive examples is realized. On every iteration, a default clause (and possibly multiple exceptions) - denoted by $C_{def+exc}$ - is learned and added to the hypothesis. Then, the covered examples are removed from the remaining examples. In the function LEARN_ONE_RULE (lines 9-17), Top-K algorithm with $k = 1$ is invoked and a high-utility itemset (i.e., a subset of features-values and their corresponding Shapley values) is retrieved. These subset of features create the default part of a new clause. Next, if the default clause covers false positives, the current positive and negative examples are swapped to learn exceptions. In the function LEARN_EXCEPTIONS (lines 18 - 25), the algorithm recursively calls itself to learn clauses that would cover exceptional patterns. When the recursive call returns, for all learned clauses, their head is replaced by an abnormality predicate. To manufacture the complete default theory, the abnormality predicate preceded by negation-as-failure (not) is added to the default part. The following example shows how SHAP-FOLD learns a concise non-monotonic logic program from an XGBoost trained model.

kir

Example 7.3. *The "UCI Cars" dataset has the information about evaluating 1728 different cars and their acceptability based on features such as buying price, maintenance cost, trunk size, capacity, number of doors, and safety. SHAP-FOLD generates the following program from a trained XGBoost model:*

Algorithm 9 Summary of SHAP-FOLD Algorithm

Input: G: Target Predicate to Learn
B: Background Knowledge
$D = \{ (\vec{x}_1, y_1), ..., (\vec{x}_m, y_m)\} : \ y_i \in \{-1, +1\}$
$E^+ = \{ \vec{x}_i \mid \vec{x}_i \in D \wedge y_i = 1\} :$ Positive Examples
$E^- = \{ \vec{x}_i \mid \vec{x}_i \in D \wedge y_i = -1\}:$ Negative Examples
$T = \{ (\vec{z}_i, \vec{\phi_i}) \mid \vec{z}_i \subseteq \vec{x}_i \wedge \vec{x}_i \in D \wedge \vec{\phi_i}$ is $\vec{z}_i$'s Shapley values $\}$
Output: $D = \{ C_1, ..., C_n\}$ ▷ default clauses
$AB = \{ ab_1, ..., ab_m\}$ ▷ exceptions/abnormal clauses

function FOIL(E^+, E^-)
  while ($|E^+| > 0$) **do**
    $C_{def+exc} \leftarrow$ LEARN_ONE_RULE(E^+,E^-)
    $E^+ \leftarrow E^+ \setminus covers(C_{def+exc}, E^+, B)$
    $D \leftarrow D \cup \{C_{def+exc}\}$
  end while
  return D, AB ▷ returns sets of defaults and exceptions
end function
function LEARN_ONE_RULE(E^+, E^-)
  - let Item-Set be $\{(f_1, ...f_n), (\phi_1, ..., \phi_n)\} \leftarrow$ TOP-K(K=1,E^+,T) ▷ Call to HUIM algorithm
  $C_{def} \leftarrow (G$:- $f_1, ..., f_n)$
  $FP \leftarrow covers(C_{def}, E^-)$ ▷ FP denotes False Positives
  if $FP > 0$ **then**
    $C_{def+exc} \leftarrow$ LEARN_EXCEPTIONS(C_{def}, E^-, E^+)
  end if
  return $C_{def+exc}$
end function
function LEARN_EXCEPTIONS(C_{def}, E^+, E^-)
  $\{C_1, ..., C_k\} \leftarrow$ FOIL(E^+, E^-) ▷ Recursive Call After Swapping
  $ab\_index \leftarrow GENERATE\_UNIQUE\_AB\_INDEX()$
  for $i \leftarrow 1$ **to** k **do**
    $AB \leftarrow AB \cup \{ab_{ab\_index}$:- $bodyof(C_i)\}$
  end for
  return $C_{def+exc} \leftarrow (headof(C_{def})$:- $bodyof(C_{def}),$ **not**$(ab_{ab\_index}))$
end function

```
DEF(1):
        acceptable(A):- safety(A,high),
                        not ab0(A).
EXCEPTIONS(1):
               ab0(A):- persons(A,2).
               ab0(A):- maintenance(A,very_high).
DEF(2):
        acceptable(A):- persons(A,4),
                        safety(A,medium),
                        not ab1(A).
EXCEPTIONS(2):
               ab1(A):- price(A,very_high),
                        trunk(A,small).
               ab1(A):- price(A,high),
                        maintenance(A,very_high).
DEF(3):
        acceptable(A):- trunk(A,big),
                        safety(A,medium),
                        persons(A,>5).
```

On first iteration, the clause DEF(1) (i.e., ***acceptable(A) :- safety(A,high)*** *is generated. Since it covers a significant number of negative examples, E^+ and E^- are swapped and algorithm recursively calls itself. Inside LEARN_EXCEPTIONS, the recursive call returns with EXCEPTIONS(1) clauses. The head predicate* ***ab0*** *replaces their head and finally in line 24, the negation of abnormality is appended to the default to create a complete default clause. According to the discovered default clause, a car is considered acceptable if its safety*

is high, unless it can only fit two person (too small) or its maintenance cost is high. Similarly, the DEF(2) clause states that a car is acceptable if it can fit 4 person and its safety is medium, unless its price is too high and its trunk is small. Another exceptional case is established by high price and very high maintenance. The third clause default part does not cover any false positives, hence no exception clause is learned.

There are some technicalities that should be pointed out: (1) The numeric features should be discretized (i.e., by splitting them into fixed number of intervals). This restriction is imposed by SHAP technique. (2) Inspired by FOIL implementation, the "if statement" in line 13 of the algorithm is realized in terms of an empirical accuracy (e.g., % 85) to avoid over-fitting. This would allow some noise error after learning the exceptions.

7.4 Experiments

In this section, we present our experiments on UCI standard benchmarks (Lichman, 2013b). SHAP-FOLD implementation is available at: `https://github.com/fxs130430/SHAP{_}FOLD`

The ALEPH system (Srinivasan, 2001) is used as a baseline. ALEPH is a state-of-the-art ILP system that has been widely used in prior work. To find a rule, ALEPH starts by building the most specific clause, which is called the "bottom clause", that entails a seed example. Then, it uses a branch-and-bound algorithm to perform a general-to-specific heuristic search for a subset of literals from the bottom clause to form a more general rule. We set ALEPH to use the heuristic enumeration strategy, and the maximum number of branch nodes to be explored in a branch-and-bound search to 500K. We also configured ALEPH to allow up to 50 false examples covered by each clause while each clause should be at least 80 % accurate. We use the standard metrics including precision, recall, accuracy and F_1 score to measure the quality of the results.

The SHAP-FOLD requires a statistical model as input to the SHAP technique. While computing the Shapley values is slow, there is a fast and exact implementation called Tree-

Table 7.2: Evaluation of SHAP_FOLD on UCI Datasets

		Algorithm									
		Aleph					SHAP-FOLD				
Data Set	Shape	Precision	Recall	Accuracy	F1	Time (s)	Precision	Recall	Accuracy	F1	Time (s)
cars	(1728, 6)	0.83	0.63	0.85	0.72	73	**0.84**	**0.94**	**0.93**	**0.89**	**5**
credit-a	(690, 15)	0.78	0.72	0.78	0.75	180	**0.90**	**0.74**	**0.84**	**0.81**	**7**
breast-w	(699, 9)	0.92	0.87	0.93	0.89	10	**0.92**	**0.95**	**0.95**	**0.93**	**2**
kidney	(400, 24)	**0.96**	0.92	0.93	0.94	5	0.93	**0.95**	0.93	0.94	**1**
voting	(435, 16)	0.97	0.94	0.95	0.95	25	**0.98**	**0.98**	0.95	**0.96**	**1**
autism	(704, 17)	0.73	0.43	0.79	0.53	476	**0.96**	**0.83**	**0.95**	**0.89**	**2**
ionosphere	(351, 34)	**0.89**	0.87	0.85	0.88	113	0.87	**0.91**	0.85	**0.89**	**2**
heart	(270, 13)	0.76	0.75	0.78	0.75	28	0.76	**0.83**	**0.81**	**0.80**	**1**
kr vs. kp	(3196, 36)	0.92	0.99	0.95	0.95	836	0.92	0.99	0.95	0.95	**8**

Explainer (Lundberg et al., 2018) for ensemble tree models. XGBoost (Chen and Guestrin, 2016) is a powerful ensemble tree model that perfectly works with TreeExplainer. Thus, we trained an XGBoost model for each of the reported experiments in this chapter. Table 7.2 presents the comparison between ALEPH and SHAP-FOLD on classification evaluation of each UCI dataset. The best performer is highlighted with boldface font. In terms of the running time, SHAP-FOLD scales up much better. In case of "King-Rook vs. King-Pawn", while ALEPH discovers 283 clauses in 836 seconds, SHAP-FOLD does much better. It finishes in 8 seconds discovering only 3 clauses that cover the knowledge underlying the model. Similarly, in case of "UCI kidney", SHAP-FOLD finds significantly fewer clauses.Thus, not only SHAP-FOLD's performance is much better, it discovers more succinct programs. Also, scalability is a major problem in ILP, that our SHAP-FOLD algorithm solves: its execution performance is orders of magnitude better.

SHAP-FOLD almost always achieves a higher Recall score.This suggests that the proper use of *negation-as-failure* leads to better coverage. The absence of negation from ALEPH hypothesis space forces the algorithm to create too specific clauses which leaves many positive examples uncovered. In contrast, our SHAP-FOLD algorithm emphasizes on better coverage via finding high-utility patterns of important features first. If the result turns out to cover too many negative examples to tolerate, by learning exceptions and ruling them out (via the

same algorithm applied recursively), SHAP-FOLD maintains the same coverage as it rules out exceptional negative examples.

SHAP-FOLD is a Java application that interfaces SWI-Prolog (Wielemaker et al., 2012) using JPL library. The HUIM instances are solved by calling TKU from the SPMF Data mining library (Fournier-Viger et al., 2016).

CHAPTER 8

CONSTRAINTS-AWARE COUNTER-FACTUAL PROPOSALS

8.1 Overview

The EU's General Data Protection Regulation (GDPR) recognizes the "right to explanation" for the decisions made by algorithms about humans. The "right to explanation" protects individuals through (1) helping them understand why an algorithm arrives at a decision (2) providing grounds to challenge the decision (3) informing the individual about what could be changed to receive a desirable decision in the future assuming that the same algorithm will be used. The latter is known as counterfactual explanation problem.

A counterfactual explanation describes a causal situation in the form "if X had resp.,(had not) the property P, event Y would resp., (wouldn't) have happened. For instance, if John, a bank customer, had an income salary greater than 100,000 $ per year, his loan application would have been approved. Alternatively, if John had 7 open accounts, his loan application would have been approved. If John's current salary is 98,000 $, a counterfactual proposal based on a 2000 $ raise in annual salary is a feasible change. However, if John's current salary is 65,000 $, a raise of 35,000 $ seems unrealistic. On the other hand, if he already has 6 open accounts, opening another account does not require too much effort on his side. As the example suggests, a solution to counterfactual explanation should always incur a minimum change made to the original feature values.

Definition 8.1. *Given an input feature vector* $x \in \mathrm{X}$*, where* X *is the set of all possible input feature vectors and a binary classifier* $f : X \to \{0, 1\}$ *which is a function that maps an input feature vector into a decision* $y \in \{0, 1\}$*, a counterfactual space* $CF_f(x)$ *of* x *is defined as follows:*

$$CF_f(x) = \{\hat{x} \in \mathrm{X} \mid f(x) \neq f(\hat{x})\}.$$

For multi-class predictors, we can always break down the problem into a binary classification instance using one-versus-rest approach. In this approach, instances from the class of interest are relabeled as "1" and instances of other classes are relabeled as "0".

Next we will characterize the Minimality property based on Definition 8.1 of counterfactual space.

Definition 8.2. *The nearest counterfactual explanation for a data sample x and a binary classifier f on a given domain* X *is defined as follows:*

$$\hat{x}^* = \underset{\hat{x} \in CF_f(x)}{\operatorname{argmin}} \; d(x, \hat{x})$$

where d is a distance function that determines how "similar" its given arguments are.

In other words, the nearest counterfactual explanation is a data point from the space of all counterfactual explanations of the data sample x characterized by $CF_f(x)$, such that it is the "closest" to the original data sample. A naive solution to generate counterfactual explanations is a trial and error search. In this approach, the entire space of counterfactual explanations (i.e., CF_f) is generated by labeling the entire set of X using function f. After that, the point with smallest distance to the original data sample is picked up as the nearest counterfactual explanation. This approach obviously does not scale up and is of no use.

Recent proposed solutions treat this problem as an optimisation problem (Wachter et al., 2017; Looveren and Klaise, 2019). The authors of (Wachter et al., 2017) suggest to minimize the following loss function:

$$L(x, x', y', \lambda) = \lambda.(f(x') - y')^2 + d(x, x')$$

In this loss function, the first term pushes the prediction $f(x')$ towards the intended outcome y' while the second term tries to minimize the distance of a counterfactual explanation x' from the original data sample x. The choice of λ parameter puts smaller or greater emphasis

on the proximity to the desired output prediction. A higher λ prefers counter-factuals that have very close desired outcome, where as, a small value of λ, prefers counter-factuals that are quite similar to the original data sample. Since choosing the λ is tricky, the authors suggest to add another constraint ϵ as the tolerance for how far away the prediction of the counterfactual is allowed to be from the desired output. This extra constraint is defined as follows:

$$|f(x') - y'| \leq \epsilon$$

To minimize the loss function for a given data sample x and a desired output y' and a tolerance parameter ϵ, first the smallest x' is found then it solves for the largest λ parameter that yields the smallest ϵ, hence it solves the following optimisation objective:

$$\underset{x'}{\operatorname{argmin}} \max_{\lambda} L(x, x', y', \lambda)$$

In (Laugel et al., 2018), authors propose an enhanced search based on growing a sphere around the given data sample and check whether any of the randomly generated data samples in the sphere fall into the boundaries of the desired class or not. If no such data point is found, the sphere grows and the same operation is repeated.

The main disadvantage of all the above approaches is that they are completely oblivious of the logical, physical and temporal constraints. For instance, if age is among the features, nothing can stop the counterfactual generation algorithm to make recommendations based on increasing / decreasing the age. Even if it does rule out some of the infeasible recommendations, the results are not guaranteed to be minimally close to the original data sample.

In this chapter, we will propose an approach based on an enhanced form of abductive Answer Set Programming that would fully respect the constraints that logic, physics and time would impose to the domain of interest and is defined as part of the background knowledge in the process of generating counter factual explanations.

8.2 Abductive Answer Set Programming

In Chapter 3, Section 3.6 we introduced abduction as a way of adding pieces of knowledge that are deemed necessary to make a certain clause hold. Adding a directive `#abducible` in s(ASP) followed by a predicate lets s(ASP) engine assume the correctness of all abductively stated predicates as it encounters them during the top down execution a query. Consider Example 8.1 and a data sample with the following set of facts:

```
safety(car12,low).                    doors(car12,4).
trunk_size(car12,big).                maintenance_cost(car12,low).
price(car12,very_high)                capacity(car12, 4).
```

Example 8.1. *The following clauses are part of the rules that were induced from UCI Car Evaluation dataset by SHAP_FOLD algorithm. UCI Car Evaluation represents the quality of different cars given their safety level, capacity, buying price, maintenance cost, trunk size and the number of doors. This is a classification task of determining whether a car has acceptable quality based on the above features.*

```
(1) acceptable(A):- safety(A,high), not ab0(A).
(2) acceptable(A):- capacity(A,4), safety(A,medium), not ab2(A).
    ab0(A):- capacity(A,2), maintenance_cost(A,high).
    ab0(A):- maintenance_cost(A,very_high).
```

The query `?- acceptable(car12).` will fail, because, `safety(car12,high)` from clause (1) and `safety(car12,high)` from clause (2) fails as car12 has `safety(car12,low)`. By adding `#abducible safety(X,high).` to the source file and running the same query again, this time it succeeds with the following Answer set:

```
{ acceptable(car12), safety(car12,high), not ab0(car12),
  not capacity(car12,2), not maintenance_cost(car12,very_high)}
```

In this example, although `safety(car12,high)` does not belong to the set of facts around `car12`, but through introducing `#abducible safety(X,high)`, the s(ASP) engine assumes that it holds and therefore, includes it in the partial Answer Set shown above. It should be noted that `#abducible safety(X,high)` is a syntactic sugar for an even loop through negation and could be replaced with the following set of clauses:

```
safety(X,high) :- not q(X).
          q(X) :- not safety(X,high).
```

where `q(X)` serves as a dummy predicate. As a result, our solution is not bound by the choice of Answer Set Programming engine, although, the partial answer set generated by s(ASP), only contains the relevant literals that are used in GL method to establish the correctness of the query. Other Answer Set Programming engines such as Clingo always ground the program first, and then output the entire least Herbrand Universe of the program. As a result they tend to be less scalable when dealing with a giant knowledge base.

Abduction *almost* solves the problem by recommending the facts that are missing from the body of a clause.These are the facts that would make the left hand side of the clause hold, However, in case of abnormality predicates, if the left hand side of a clause does not hold, abduction mechanism cannot help, as it can only find the missing facts, but in case of abnormalities, something holds that should not have. For clarification, consider the following example from another car instance in UCI Car Evaluation domain:

```
safety(car14,low).                    doors(car14,4).
trunk_size(car14,big).                maintenance_cost(car14,very_high).
price(car14,very_high)                capacity(car14, 4).
```

In this case, abduction cannot satisfy clause (1).
This is because, `maintenance_cost(car14,very_high)` satisfies the exception clause `ab0`. The only way to satisfy clause (1) is to remove `maintenance_cost(car14,very_high)` from

the set of facts. Therefore, to handle both addition and subtraction of facts, a stronger mechanism is needed. One that is capable of (a) finding the missing facts (b) removing the facts that through negation-as-failure make the left hand side of the target clause false (c) being able to handle arithmetic constraints and inequalities. This mechanism leverages the Craig Interpolants and will be discussed in the next section.

8.3 Craig Interpolants

Craig's interpolation theorem (Craig, 1957) in mathematical logic states that if $A \implies C$ is a valid (closed) implication in first-order logic, then there is a Craig interpolant I such that $A \implies I$ and $I \implies C$ are valid and every non-logical symbol of I occurs in both A and C. "Non-logical" symbols are variables and uninterpreted functions and so on. There is a reverse form of this theorem with important applications in model checking that is defined below:

Definition 8.3. *For two subsets of clauses (A, B) such that $A \wedge B$ is unsatisfiable, there is a reverse interpolant I such that $A \implies I$ and $B \implies \neg I$ and every "non-logical" symbol of I occurs in both A and B.*

Reverse interpolant for $A \wedge B$ is identical with ordinary interpolant for $A \implies \neg B$. Hence, from this point and onward, whenever interpolants come up, we really mean the reverse interpolants. In Definition 8.3, a clause is a disjunction of zero or more positive or negative propositional variables that are not tautological. In other words, no clause contains a variable and its negation. Given two clauses of the form $c_1 = p \vee A$ and $c_2 = \neg p \vee B$, a resolvent of c_1 and c_2 is the clause $A \vee B$ as long as $A \vee B$ is not a tautology.

Definition 8.4. *A proof of unsatisfiability P for a pair of clauses C is a DAG (directed acyclic graph) with (V_p, E_p) where V_p is a set of clauses, such that for every vertex $c \in V_p$:*

- *c is a leaf and* $c \in C$
- *c is the resolvent of exactly two parents* C_1 *and* C_2
- *c is the empty clause (as a result of resolving* $p \wedge \neg p$

Mcmillan in (McMillan, 2003) proposed a systematic way of computing interpolants for two pairs of inconsistent clause sets (A, B) using the resolution proof of unsatisfiability P of $A \cup B$. A literal is *global* if it occurs in both A and B, and it is *local* otherwise. For any clause c, $g(c)$ ($l(c)$) denotes a disjunction of all its global (local) literals respectively.

Example 8.2. *For the pair of clause sets* (A, B) *such that* $A = \{\bar{b}, \bar{a} \vee b \vee c, a\}$, $B = \{\bar{a} \vee \bar{c}\}$ *we have* $g(\bar{b}) = \bot$ *because the variable b is not global (only occurs in A). Also, we have* $g(\bar{a} \vee b \vee c) = \bar{a} \vee c$.

Definition 8.5. *Let* (A, B) *be a pair of clause sets and P be a proof of unsatisfiability of* $A \cup B$*, with the root (bottom) vertex r. For all vertices* $c \in V_p$ *let* p_c *be a boolean formula named a partial interpolant such that:*

- *if c is a leaf, then*
  - *if* $c \in A$ *then* $p(c) = g(c)$,
  - *else* $p(c)$ *is constant* $True$
- *else, let* c_1, c_2 *be the predecessors of c and let v be the resolved variable*
  - *if v is local, then* $p(c) = p(c_1) \vee p(c_2)$,
  - *else* $p(c) = p(c_1) \wedge p(c_2)$

The interpolant of two inconsistent clause sets (A, B) *is* p_r *where r is the root of the proof DAG (bottom clause). Interpolant is denoted by* $ITP(A, B)$.

Figure 8.1 shows the annotated resolution proof tree of the pair of clauses sets from Example 8.2. The $ITP(A, B)$ is the corresponding partial interpolant of bottom clause.

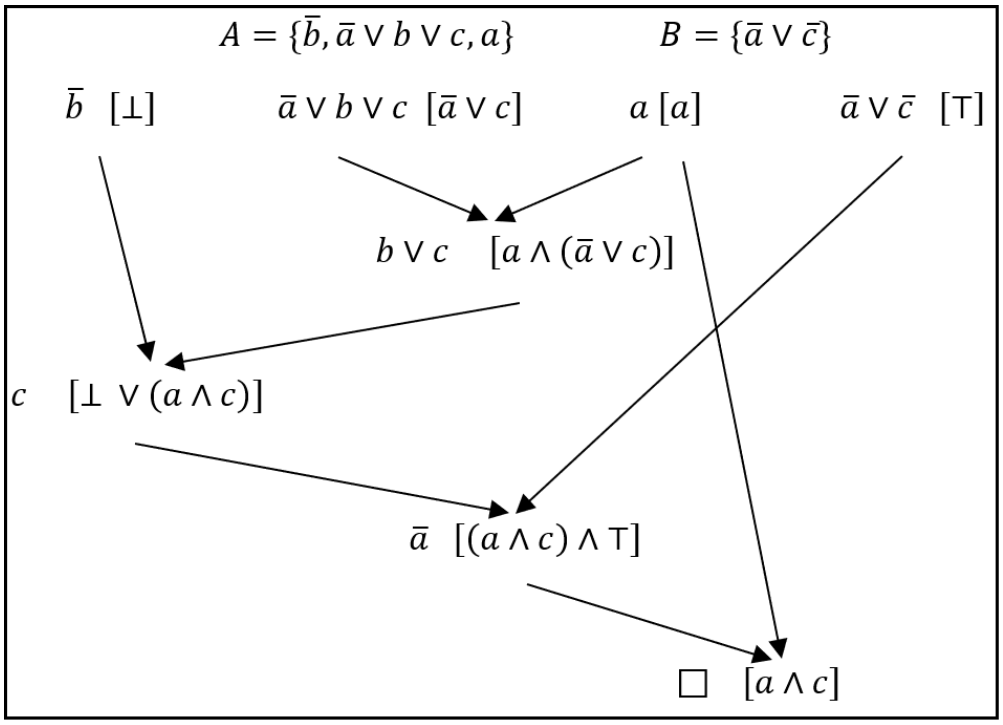


Figure 8.1: Computing the Craig interpolant for two sets of inconsistent clauses using the resolution proof tree annotations

8.4 ASP-based Counterfactual Explanation Using Craig Interpolants

In Section 8.2 We showed how by defining a predicate as `#abducible` (or alternatively, by introducing an even loop though negation-as-failure) the ASP engine can fill out the missing facts as it encounters them. In simple terms, every #abducible creates two worlds. In one world, the abduced predicate is considered to hold and in the other world it is considered to be false. Therefore, if all the body predicates of our hypothesis are defined as abducible predicates, s(ASP) engine will systematically create all possible paths in the program to satisfy the hypothesis. s(ASP) will represent each of these paths using a partial answer set that contains all relevant abductive facts necessary to satisfy the hypothesis using that path. On the other hand, we have a set of facts representing a data sample's feature values. In the context of counter-factual explanation the assumption is that none of the paths to satisfy the hypothesis are consistent, because otherwise, the data sample would not need a counter-

factual explanation on the first place. Therefore, each of the paths to satisfy the hypothesis, that is, each of the answer sets generated by s(ASP) engine, provide a proposal from CF_f space to counter-factually explain the current data sample. Since each answer set is a set of formulae, that are inconsistent with the set of facts from our data sample, Craig interpolant of the two sets can be computed. The interpolant explains what makes the hypothesis and a given data sample inconsistent. Hence, it determines what needs to change in the original data sample to satisfy the hypothesis. By repeating the same process for each possible path, a counter-factual proposal is created. It should be noted that every path provides a data point from the counter-factual explanation space, however, to find the nearest one to the original data sample, we still need to define an appropriate distance function that compares each counter-factual proposal with the original data sample and picks up the "closest" one.

Example 8.3. *Listing 1 shows a partial set of clauses that were learned from the UCI Car Evaluation data set in the previous chapters. The following shows some of the partial answer sets generated by s(ASP):*

```
{ acceptable(id,1), safety(id,high), not ab0(id), not capacity(id,2),
not maintenance_cost(id,very_high) }
```

This partial answer set shows that one way of satisfying the first clause (i.e., `acceptable(id,1)`*) is to establish* `safety(id,high)` *while making sure that both* `ab0` *clauses fail. This is established via* `not capacity(id,2)` *from the first clause of* `ab0` *and* `not maintenance_cost(id,very_high)` *from the second clause of* `ab0`*.*

```
{ acceptable(id,1), capacity(id,2), safety(id,high), not ab0(id),
not maintenance_cost(id,high), not maintenance_cost(id,very_high) }
```

This partial answer set allows `capacity(id,2)` *to hold, instead, it fails the* `ab0` *goal by including* `not maintenance_cost(id,2)` *in the partial answer set.*

Listing 1 UCI Car Evaluation First Two Clauses

```
% second argument is used to determine which rule is being used
acceptable(A,1):- safety(A,high), not ab0(A).
acceptable(A,2):- capacity(A,4), safety(A,medium), not ab2(A).

ab0(A):- capacity(A,2), maintenance_cost(A,high).
ab0(A):- maintenance_cost(A,very_high).

ab2(A):- price(A,very_high), trunk_size(A,small).
ab2(A):- price(A,high), maintenance_cost(A,very_high).
ab2(A):- maintenance_cost(A,very_high), price(A,very_high).
ab2(A):- price(A,very_high), maintenance_cost(A,high).
ab2(A):- price(A,high), trunk_size(A,small).

#abducible safety(A,X).
#abducible capacity(A,X).
#abducible maintenance_cost(A,X).
#abducible trunk_size(A,X).
#abducible price(A,X).
#abducible doors(A,X).

% Compute All Answer sets with switch 0
#compute 0 {acceptable(id,N)}.
```

```
{ acceptable(id,2), capacity(id,4), safety(id,med), not ab2(id),
not price(id,high), not price(id,very_high),
not maintenance_cost(id,very_high) }
```

This partial answer set shows one way of satisfying the second clause. Each of the negated literals contribute to the failure of one of `ab2` *clauses.*

After generating all partial answer sets, Craig-interpolant is leveraged to create counterfactual proposals. To achieve this, a pair (A, B) of clause sets is created with the partial answer set as A and the set of facts from a given data sample as B. However, it should be noted that for categorical features, one-hot encoding is performed first, and for each feature value with value 0, a negated fact is also added to the set B. As an example we have the facts

related to `car12` along with the first partial answer that s(ASP) generates for the hypothesis in Listing 1.

```
A = { acceptable(car12,1), safety(car12,high), not capacity(car12,2),
      not maintenance_cost(car12,very_high) }
B = { safety(car12,low), doors(car12,4), trunk_size(car12,big),
      maintenance_cost(car12,low), price(car12,very_high),
      capacity(car12, 4), not capacity(car12, more_than_5),
      not capacity(car12,2), not maintenance_cost(car12, high),
      not maintenance_cost(car12,very_high), not doors(car12,2),
      not doors(car12,3), not trunk_size(car12,small),
      not trunk_size(car12,medium), not safety(car12, low),
      not safety(car12, medium), not price(car12,high),
      not price(car12,medium), not price(car12,low)}
```

The Craig interpolant for the pair (A, B) is as follows:

$ITP(A, B) =$ `not capacity(car12,2) AND not maintenance_cost(car12,very_high)`

To make the interpolant affirmative instead of prohibitive, each negated literal can be replaced by the disjunction of the rest of possible values. In case of Example 8.3, the counter-factual proposal after removing the negated literals looks like the following:

1. capacity $\in \{4,\ more\_than4\} \wedge$ maintenance_cost $\in \{high,\ low\}$

2. maintenance_cost = low

The rest of counter-factual proposals are generated using the second clause, that is `acceptable(id, 2)` as follows:

1. price $\in \{low, medium\} \wedge$ maintenance_cost $\in \{high, low\} \wedge$ safety = medium

2. price $\in \{low, medium\} \wedge$ capacity $= 4 \wedge$ safety $=$ medium

3. price $=$ high $\wedge$ capacity $= 4 \wedge$ safety $=$ medium $\wedge$ maintenance_cost $\in \{low, high\}$

4. capacity $= 4 \wedge$ maintenance_cost $=$ low $\wedge$ safety $=$ medium

5. price $=$ high $\wedge$ capacity $= 4 \wedge$ maintenance_cost $=$ low $\wedge$ safety $=$ medium

Algorithm 10 summarizes the logic-based approach presented in this chapter to find the constraint-aware counter-factual explanation proposals. In this algorithm, all the constraints are presented as part of the background knowledge and contribute to the abductive answer sets that are produced. In Algorithm 10, $ITP(as, x)$ denotes the process of computing Craig interpolant by having as as A and the set of facts around the data sample x as B. Also, the function *apply* on line 10, creates a new copy of the data sample with all the facts from interpolant applied to the original data sample to create the counter-factual proposal x'. Next, we prove that our Interpolating CFE algorithm's proposals always result in flipping the classification decision (soundness).

Theorem 8.1. *The Interpolating CFE algorithm is sound.*

Proof. By contradiction: Assume for the sake of contradiction that CFE algorithm is not sound. Thus, it generates at least one counter-factual proposal c for data sample x that does not belong to the desirable class. Without loss of generality, we assume that a path p with the answer set AS was used to generate c. Since c does not belong to the desirable class, it follows that c is not implied by the path p, hence, the answer set AS and c are inconsistent. This contradicts the definition of an interpolant, because, AS must imply the interpolant, that is, $AS \implies c$. □

Next, we prove that if a counter-factual proposal exists, our interpolating CFE algorithm will find it (completeness)

Algorithm 10 Summary of Interpolating CFE Algorithm

Input: $D = \{ C_1, ..., C_n\}$ ▷ default clauses
$AB = \{ ab_1, ..., ab_m\}$ ▷ exceptions' clauses
B: Background Knowledge
x: Set of facts around data sample x
Output: $E = \{ e_1, ..., e_p\}$ ▷ Counter-factual proposals

1: **function** FIND_COUNTER_FACTUALS(x)
2: **for each** $c_i \in D$ **do**
3: **for each** $p_j \in c_i$ **do**
4: Define p_i as #abducible
5: **end for**
6: **end for**
7: Run s(ASP) on the program $D \cup AB \cup B$ to get set AS of answer sets
8: **for each** $as \in AS$ **do**
9: $ip = ITP(as, x)$
10: $x' = apply(ip, x)$
11: $E = E \cup \{x'\}$
12: **end for**
13: **return** E
14: **end function**

Theorem 8.2. *The Interpolating CFE algorithm is complete.*

Proof. By contradiction: for the sake of contradiction, we assume that there is a path p through a clause cl that is not covered by our Interpolating CFE algorithm. By defining every predicate pr on the bodies of hypothesis's clauses, for each predicate, an even loop through negation-as-failure is created as follows:

```
pr :- not q.

q  :- not pr.
```

where `q` is a dummy predicate. According to the GL method, this produces two branching factor, one with predicate pr and one without it. Equivalently, this abductive definition yields two answer sets: The first answer set includes `pr` and the second answer set includes `not pr`. If the path p is not satisfied, there exists a predicate `pr` for which neither the

predicate nor its negation was used. This contradicts the GL method in the sense that either `pr` or its negation must always appear in all answer sets. □

To compute the Craig interpolants, Princess SMT solver (Rümmer, 2008) is used. Mcmillan in his seminal work (McMillan, 2004) extends the interpolants from the proofs of unsatisfiability. In particular he proposes annotation systems to compute interpolants for theories with real numbers, inequality constraints and uninterpreted functions. In the next section, we also extend our Interpolating CFE algorithm over hypotheses with inequality constraint over real numbers.

8.5 Interpolating CFE Algorithm For hypotheses with arithmetic constraints

Abductive answer set programming cannot handle arithmetic constraints. Therefore, we first need to transform them into an ASP friendly representation. Arithmetic constraints are usually of the following form in logic programming:

$$feature\_value(ID, N), N\ op\ \phi$$

where ID is the identifier variable with a feature_value N that N must be constrained by operator op to the threshold ϕ and $op \in \{<, >,\ \leq,\ \geq,\ ==\}$.

Example 8.4. *Inspired by the UCI Car Evaluation dataset and Example 8.1, we assume that a car mileage irrespective of other features should be always less than or equal 10,000 miles. Listing 2 reflects the necessary changes made to the hypothesis from Listing 1 in order to add mileage constraint to the hypothesis.*

In this example, `mileage(A,1000,ste)` *denotes the constraint that mileage of a car with id equals A should be smaller than or equal (i.e., ste) 10000.*

Once the normal arithmetic constraints are replaced by the transformed predicates, they can be treated using the same process by our Interpolating CFE Algorithm. However, in

Listing 2 UCI Car Evaluation First Two Clauses

```
% second argument is used to determine which rule is being used
acceptable(A,1):- mileage(A,1000,ste), safety(A,high), not ab0(A).
acceptable(A,2):- mileage(A,1000,ste), capacity(A,4), safety(A,medium),
                  not ab2(A).

ab0(A):- capacity(A,2), maintenance_cost(A,high).
ab0(A):- maintenance_cost(A,very_high).

ab2(A):- price(A,very_high), trunk_size(A,small).
ab2(A):- price(A,high), maintenance_cost(A,very_high).
ab2(A):- maintenance_cost(A,very_high), price(A,very_high).
ab2(A):- price(A,very_high), maintenance_cost(A,high).
ab2(A):- price(A,high), trunk_size(A,small).

#abducible safety(A,X).
#abducible capacity(A,X).
#abducible maintenance_cost(A,X).
#abducible trunk_size(A,X).
#abducible price(A,X).
#abducible doors(A,X).
#abducible mileage(A,X,OP).

% Compute All Answer sets with switch 0
#compute 0 {acceptable(id,N)}.
```

order to compute the interpolant from the answer set, all the transformed predicates are converted back to their normal form. The Listing 3 shows the Princess SMT code in order to compute the interpolant for the following answer set generated by the first clause of the program from Listing 2 for `car12` with mileage 11000.

```
{ acceptable(id,1), mileage(id,10000,ste), safety(id,high), not ab0(id),
not capacity(id,2), not maintenance_cost(id,very_high) }
```

Listing 3 Princess Code for Computing Craig Interpolant

```
\functions {
bool trunk_small, buying_medium, doors_morethan4, trunk_medium, safety_low;
bool price_high, price_very_high, safety_high, capacity_4, capacity_2, doors_4;
bool maintenance_cost_very_high, doors_2, doors_3, trunk_big;
bool capacity_morethan4, maintenance_cost_low, price_low;
bool safety_medium, maintenance_cost_high;
int mileage;
}
\problem {
  \part[left] ( mileage <= 10000 & safety_high &
                !capacity_2 & !maintenance_cost_very_high) &
  \part[right] ( price_very_high & !price_high & !price_medium &
                 !price_low & maintenance_cost_very_high &
                 !maintenance_cost_low & !maintenance_cost_high & doors_3 &
                 !doors_2 & !doors_4 & !doors_morethan4 & capacity_2 &
                 !capacity_4 & !capacity_morethan4 & trunk_big &
                 !trunk_small & !trunk_medium & safety_high &
                 !safety_low & !safety_medium & mileage = 11000)
                 -> false
}
\interpolant {left; right}
```

CHAPTER 9

A FULLY EXPLAINABLE FRAMEWORK TO HANDLE VISUAL QUESTION ANSWERING TASKS

9.1 Acknowledgement

The paper (Basu et al., 2020) which forms the basis for this chapter was co-authored by Kinjal Basu, myself, and Dr. Gopal Gupta. Initially, I came up with the idea of proposing a framework that extracts the knowledge by neural networks, transforms the questions into ASP programs, and finally, performs ASP based reasoning to come up with an answer. I also, developed the neural network component. However, the implementation of NLP and reasoning components are due to Kinjal. The published paper was recognised as the second best paper in the PADL 2020 conference. We would not have got this far, if it was not for the hard work and dedication of Kinjal.

9.2 Overview

Visual Question Answering is the task of answering a question given in natural language about an image that is also given as input. Visual Question Answering has been a long standing goal of research in Artificial Intelligence. In recent years, there has been a surge in Visual Question Answering (VQA) systems based on different Neural Network architectures. For example, Stacked Attention Networks (CNN + LSTM + SA) (Yang et al., 2015), Relation Networks (CNN + LSTM + RN) (Santor et al., 2017), and Feature-wise Linear Modulation (CNN + GRU + FiLM) (Perez et al., 2018) combine the CNN-extracted image features with LSTM-extracted question features and pass them through multi-layer perceptron network. N2NMN (Hu et al., 2017), Dependency Tree (Cao et al., 2018) and TbD+reg+hres (Mascharka et al., 2018) assemble a graph of trained neural modules on the fly, each responsible for performing a single unit of computation to answer a question.

IEP (Johnson et al., 2017), DDRprog (Suarez et al., 2018) and NS-VQA (Yi et al., 2018) construct intermediate functional units that unlike N2NMN are handcrafted programs. The latter incorporates segmentation techniques to achieve more accurate vision results. MAC (Hudson and Manning, 2018) proposes differentiable reasoning units of recurrent neural network that would decompose the reasoning task to multiple small steps. While in some tasks, many of these systems have been very competitive in terms of their ability to find the correct answer, they suffer from fundamental deficiencies that dramatically restricts their applicability: The end-to-end architectures based on convolutional Neural Networks and (Long Short Term Memory) LSTMS are not explainable, they are notoriously hard to train, there is no way to incorporate any form of common-sense background knowledge in these networks, any expansion of knowledge requires the entire system to be re-engineered and trained from the scratch, and last but not the least, end-to-end neural networks work completely based on pattern-matching,whereas, we are more interested in a truly intelligent system that understands the question and finds the answer based on a chain of reasoning.

In this chapter, we propose a AQuA,a framework based on answer set programming to handle the Visual Question Answering (VQA) task. AQuA incorporates an off the shelf natural language dependency parser (Stanford CoreNLP) to transform a question into an answer set program. This program, along with a set of facts and relations that are extracted from an image using a neural network model offers a natural way of solving the visual question answering task. In this approach, the use of machine learning is restricted to the task of computer vision and understanding the question and reasoning about it is achieved through the logic programming engine.

To develop Visual Question Answering machine learning models, a numerous datasets has been created (Gao et al., 2015; Krishna et al., 2017; Malinowski and Fritz, 2014; Yu et al., 2015; Ren et al., 2015; Shah et al., 2019). However, as Johnson et. al. describes in (Johnson et al., 2017), machine learning models tends to develop "cheating" methods

by exploiting the correlations between word occurrences, rather than truly understanding the semantics behind the question. CLEVR (Johnson et al., 2017) on the other hand, is a data set of complex questions over the spatial relationships between the rendered 3d objects in an image. The complexity and diversity of questions, keeps machine learning models from developing cheating capabilities to answer questions. While the recognition and localization of objects has been made possible thanks to the advancements in convolutional neural networks, to truly understand and reason about the relationships among the objects, an intelligent system needs to leverage logic and to perform complex chain of reasoning. This latter issue, motivated us to pick up CLEVR data set and implement our AQuA framework using it.

9.2.1 YOLO - Object Detection & Localization

The YOLO architecture proposed by Redmon et. al. (Redmon et al., 2016) was a turning point in the sense that it made the near real-time detection and localization of objects in an image possible. Prior to YOLO, all neural network-based object detection and localization worked based on sliding a window over the entire image and getting the network to predict the portions of image that was surrounded by the sliding window. At the end, the windows with the highest prediction probabilities would be selected. The problem with this approach was that it would result in a huge number of sliding windows and would potentially take a long time before the final result could be extracted.

In contrast, YOLO combines the classification and regression tasks together by only using the extracted features from deep convolutional layers to solve the regression task of finding object boundaries in a single step. To scale up the naive sliding window approach, YOLO divides the image to a grid of $N \times N$ cells. Each cell has a number of predetermined boxes with different sizes (anchor boxes). During the network training through back-propagation, only the promising anchor boxes are used in minimizing the loss objective. These are the boxes with an acceptable overlap with the ground truth boxes.

9.2.2 Stanford CoreNLP Dependency Parser

Stanford CoreNLP is an off-the-shelf set of Natural Language Processing tools (Manning et al., 2014). In this work, we primarily use the *Parts of Speech* (POS) tagger and the dependency parser. Once the POS tagger tags each word in the sentence, the dependency parser finds the syntactical relationships between different words. The figure 9.1

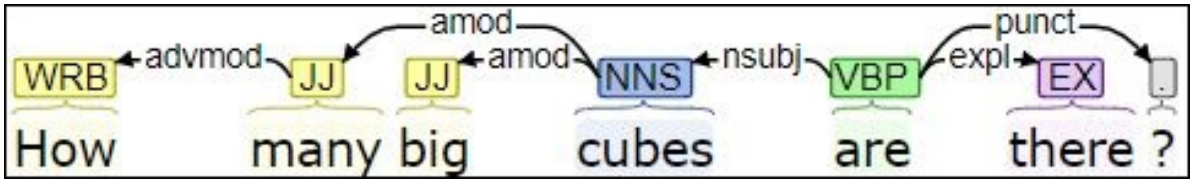


Figure 9.1: Example of POS tagging and dependency graph

9.3 The Technical Approach

The AQuA framework consists of 5 modules to perform the following tasks:

1. Object Detection and feature extraction using YOLO model
2. pre-processing of the natural language question
3. semantic relation extraction from the question
4. ASP Query generation based on semantic analysis
5. common-sense knowledge representation

Figure 9.2 summarizes the AQuA architecture. The 5 aforementioned modules are as follows: YOLO model, Preprocessor, Semantic Relation Extractor (SRE), ASP Query Genrator, and Common-sense knowledge.

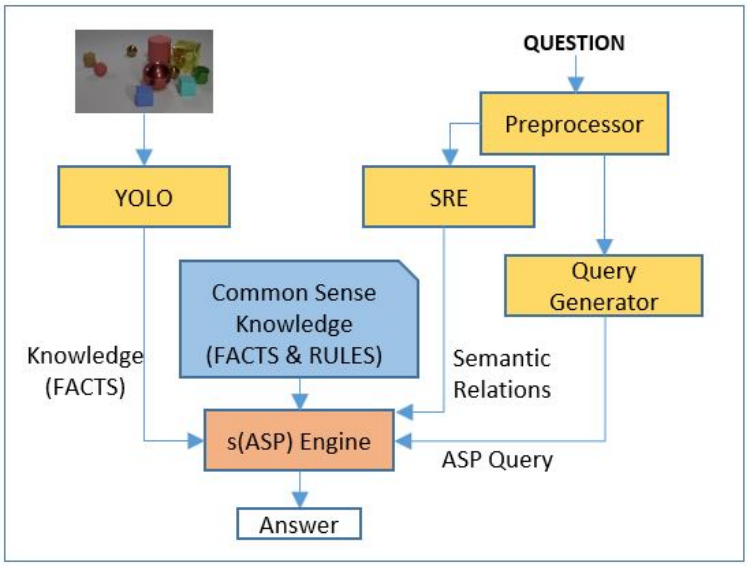


Figure 9.2: System Architecture

9.3.1 Preprocessor

Using the *parts-of-speech* tags, word lemmas and the dependency graph pf the sentence, AQuA translates a question into a sequence of predicates that would form an ASP query. Additionally, preprocessor determines the question type. This is important because, the answer type depends on the question type. For instance, the answer of a "how many" question is a numeric, the answer of an "is there / are there" question is a boolean an so on. The output of preprocessor module will be consumed by the Semantic Relation Extractor (SRE) module and the Query Generator module.

Translation of a Natural Language Question to an ASP query in AQuA is inspired by Neo-Davidsonian formalism (Davidson, 2001), where every verb is recognised as a unique event whose occurrence is assigned a unique identifier. Every word in the sentence has an identifier identical to their positional index. Therefore, even if the *queried object* and the `referenced object` have the same name, they are distinguishable. For instance, in the question "How many matte blocks are behind the red block?", the queried object and the referenced object are not the same.

9.3.2 Semantic Relation Extractor (SRE)

Semantic Relation labeling is the task of assigning a relationship label to two different phrases in a sentence based on the context. AQuA, primarily introduces the following two types of semantic relationships:

- **Quantification:** AQuA, treats all existential questions as a special type of numeric comparison. For example, `quantification(1,cube_4)` where 4 is the identifier of the word cube from the sentence), is extracted from the question "Are there any cubes?". Although, it is beyond the scope of question asked in CLEVR dataset, it helps to answer a question such as "Are there more than five objects?". To answer a question, the number argument from the quantification predicate is compared against the number of objects in a list that is passed through a list of criteria extracted from the question.

- **Property:** In CLEVR domain, objects are preceded with none, one or many attribute values. Their parts-of-speech tag is adjective. These adjectives effectively create filters for the set of objects in an image. Therefore, to capture these criteria from the questions, AQuA, represents them as `property(value, object)` predicate. For example, in the sentence "Is there a big red ball?", AQuA extracts `property(big_4, ball_6)` and `property(red_5, ball_6)` semantic relations and adds them to the knowledge base.

9.3.3 Query Generator

For a given question, AQuA generates a list of ASP clauses that once executed using the s(ASP) engine, it will find the answer. There are two types of single-word response questions: (i) yes / no questions (ii) attribute/value questions. CLEVR question set has yes/no questions in the form of existential questions (e.g., "Is there a red ball?") and value comparison questions (e.g., "Is the ball same color as the cube?"). The attribute/value questions are in

the form of counting objects (e.g., "How many red balls are there?") and querying object attributes (e.g., "What is the color of the shiny cylinder?"). To handle each type of question, AQuA generates a set of ASP clauses. For instance, to answer the yes/no question "Is there a big ball?", the following set of clauses are generated:

```
1) query(Q,A) :- question(Q), answer(A).
2) answer('is there a big ball?',yes):- find_ans('is there a big ball?').
3) answer('is there a big ball?',no):- not find_ans('is there a big ball?').
4) find_ans(Q):- question(Q), find_all_filters(ball,5,L),
                 list_object(L, Ids), list_length(Ids, C),
                 quantification(N, ball_5), gte(C, N).
5) question(‘is there a big ball ?’).
```

In this code snippet, clause (1) stores the answer to the question Q in variable A. If clause (2) succeeds, the answer to the question is yes, otherwise, as specified in clause (3), if the predicate `find_ans` fails, the answer is no. If all the sub-goals stated in the body of clause (4) succeed, then the predicate `find_ans` succeeds. Finally, fact (5) represents the natural language question.

For the attribute/value questions the following set of clauses are generated for the question "What color is the cube?":

```
1) query(Q, A) :- question(Q), answer(Q, A).
2) answer(‘what color is the cube ?’, A) :-
                    find_ans(‘what color is the cube ?’, A).
3) find_ans(Q, A) :- question(Q), find_all_filters(cube, 5, L),
                    list_object(L, Ids), get_att_val(Ids, color, A).
4) question(‘what color is the cube ?’).
```

clause (1) and (4) are similar to the clauses (1) and (5) for yes/no questions template above. Clause (2) stores the answer in variable A and carries it around to clause (1). In clause (3) filters are applied to the list of images in the scene and stores the answer in variable A.

9.3.4 Commonsense Knowledge

AQuA requires the common-sense knowledge about features (e.g., color, size, material), spatial relationships (e.g., left, front, behind), and shapes (e.g., sphere, cube, cylinder). It also needs to know for instance that, red is a color, cube is a shape, metal is a material and so on. Also, a deeper common-sense knowledge is required to infer from "shiny object" a reference to metal.

The following types of common-sense knowledge is added along with the questions:

- **Common-sense Facts:** CLEVR incorporates two types of facts: (i) attribute values (e.g., red is a color, cube is a shape) which are represented as `is_property(V, A)` as in `is_property(red, color)` and (ii) term similarities (e.g., block is similar to cube) which are represented as `is_similar(X1,X2)` as in `is_similar(big, large)`

- **Common-sense Rules:** These are the rules that would perform common sub-tasks such as list union, numeric comparison, list element counting and so on. The following code snippet shows an the set of clauses to filter a list of objects that match a specific attribute / value pair. The code is a typical recursive iteration over a list:

```
(1) filter(_, _, [], []).
(2) filter(Att, Val, [Id | T1], [Id | T2]) :- property(Id, Att, Val),
                                               filter(Att, Val, T1, T2).
(3) filter(Att, Val, [Id | T1], T2) :- not property(Id, Att, Val),
                                        filter(Att, Val, T1, T2).
```

Table 9.1: Question type wise summarized result from various state-of-the-art neural-network based model for CLEVR

Method	Count	Exist	Compare Number	Compare Attribute	Query Attribute	Overall
Humans (Johnson et al., 2017)	86.7	96.6	86.4	96.0	95.0	92.6
CNN+LSTM+SAN	59.7	77.9	75.1	70.8	80.9	73.2
N2NMN	68.5	85.7	84.9	88.7	90.0	83.7
Dependency Tree	81.4	94.2	81.6	97.1	90.5	89.3
CNN+LSTM+RN	90.1	97.8	93.6	97.1	97.9	95.5
IEP	92.7	97.1	98.7	98.9	98.1	96.9
CNN+GRU+FiLM	94.5	99.2	93.8	99.0	99.2	97.6
DDRprog	96.5	98.8	98.4	99.0	99.1	98.3
MAC	97.1	99.5	99.1	99.5	99.5	98.9
TbD+reg+hres	97.6	99.2	99.4	99.6	99.5	99.1
NS-VQA	99.7	99.9	99.9	99.8	99.8	99.8

9.4 Experiments and Results

Unlike deep-learning based approaches, AQuA is always bound to find the correct answer, as long as the information coming from various sources, including the YOLO object detection and localization module, and dependency parser are correct. While, AQuA achieves competitive results compared to the human base-line performance(i.e., 92.6 %), there are neural-network based approaches that achieve results with 99 percent accuracy. Table 9.1 summarizes the performance of some of these neural-based end-to-end systems.

The main advantages of the AQuA approach, over neural, end-to-end learning is that one can always pin-point the error source and fully explain why AQuA arrives at a certain answer for a given question. Not to mention that there is no training as far as the inference and translating of the textual question into logic are concerned. We ran AQuA on 45,157 questions that satisfy the selection criteria imposed (e.g., questions with 15 words or less) to generate ASP queries. An accuracy of 93.7 % was achieved with 42,314 correct answers. This performance is beyond the average human accuracy (Johnson et al., 2017). We validated our AQuA framework using the validation data portion of CLEVR that contains 149991

Table 9.2: Performance Results

Question Type		*Accuracy (%)*	
Exist		**96**	
Count		**91.7**	
Compare Value	Shape	87.42	**92.89**
	Color	94.32	
	Size	92.17	
	Material	96.14	
Compare Integer	Less Than	97.7	**98.05**
	Greater Than	98.6	
	Equal	NA [1]	
Query Attribute	Shape	94.01	**94.39**
	Color	94.87	
	Size	93.82	
	Material	94.75	

questions over 15000 images. While theoretically, we could cover the entire set of questions, we simplified the process by limiting the question length to 15 words.

We have extensively studied the 2,843 questions that produced erroneous results. 2,092 questions out of 2,843 do not match the correct answer and other 751 questions throw ASP exceptions. Our manual analysis showed that mismatch happens mostly because of errors caused by the YOLO module: failing to detect a partially visible object, wrongly detecting a shadow as an object, wrongly detecting two overlapping objects as one, etc. Eliminating these errors through manual intervention resulted in another 2,626 questions out of the 2,843 questions being answered correctly. Only 217 incorrectly answered questions remained. Further analysis indicated that these could be attributed to wrong parsing or oversimplified spatial reasoning. As an example of parsing error, *block* sometime is parsed as a *verb* instead of a *noun*. With respect to oversimplification of spatial reasoning, note that objects in CLEVR have 3D shapes, but we only considered X and Y coordinates to calculate relative positioning of referenced objects (e.g., for *behind the block* concept). Pinpointing the source of errors is an advantage of AQuA over all end-to-end approaches, as one could mitigate the

[1]Equality questions are minuscule in number so currently ignored

errors once their exact source is realized. For instance, instead of using a single dependency parser, we can have a triple redundant architecture with three different parsers and go by majority vote. Quantitative results for each question type are summarized in Table 9.2.

In next section we will discuss a complete example and illustrate how AQuA will find the answer through transforming a natural language question, image features and common-sense knowledge into an ASP program that is queried in s(ASP) to find the answer.

9.4.1 A Complete Example

In this section, a complete Visual Question Answering example and the entire data pipeline and inference steps are discussed. Figure 9.3 shows a scene with 3 different objects. The boundary boxes and all characteristics that are extracted by YOLO algorithm are also shown. The number at the end is the confidence score of our neural network about the detected object.

Object Representation: For each object, extracted information are also encoded as ASP facts in the following form where every object has an identifier, shape, color, material, size and coordinates in that order:

```
object(1, cylinder, cyan, rubber, small, 246, 185).
object(2, cube, red, metal, small, 270, 130).
object(3, cube, gray, metal, small, 79, 191).
```

Question: *Is there a matte thing in front of the metallic thing behind the gray cube?*

Figure 9.4 shows the parts-of-speech tagging and the dependency parser's output.

Semantic Relations: The following semantic relations are extracted from the dependency graph using default ASP rules.

```
quantification(1, thing_5).
property(matte_4, thing_5).
```

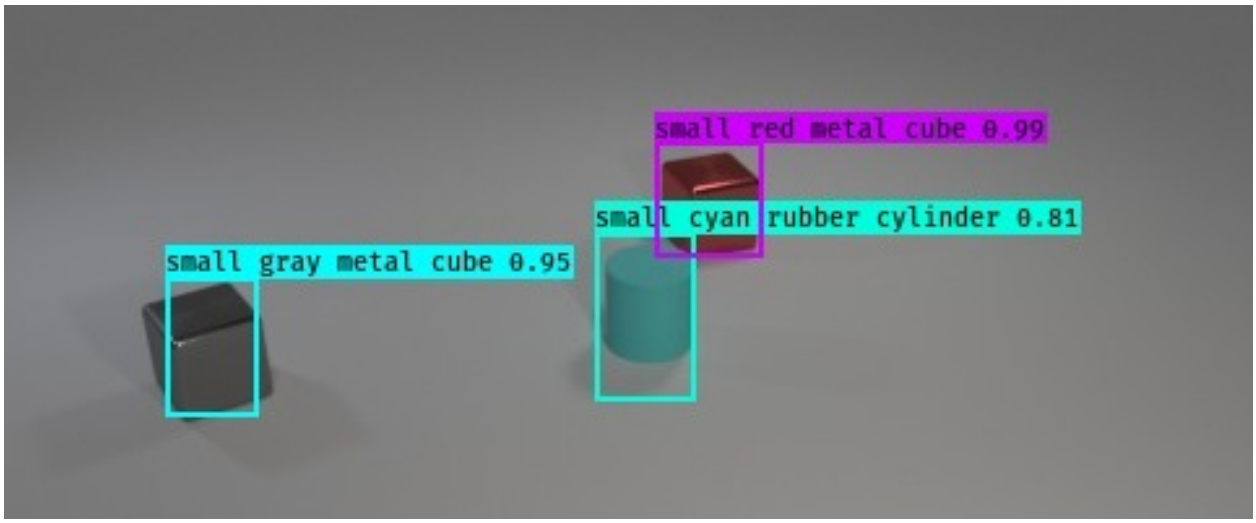


Figure 9.3: Object detection using YOLO.

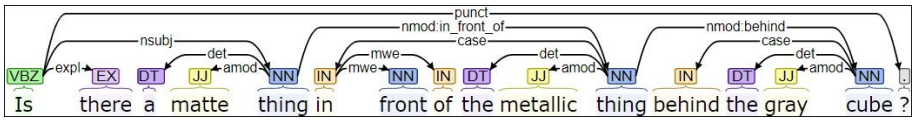


Figure 9.4: POS tagging and dependency graph

```
property(metallic_10, thing_11).
property(gray_14, cube_15).
```

Each word's lemma is followed by its positional index. The lemma "thing" occurs twice in the question where each occurrence represents a different object.

Common-sense Knowledge The following set of facts are required to understand this question:

```
is_property(cube, shape).
is_property(cylinder, shape).
is_property(metal, material).
is_property(rubber, material).
is_property(small, size).
is_property(red, color).
```

```
is_property(cyan, color).
is_property(gray, color).
is_similar(matte, rubber).
```

Query: To answer this question, a human would start from the last object (i.e., cube) and make her way back to the beginning. Similarly, AQuA, starts from the cube.

`find_all_filters(cube, 15, L2)` computes a list of all properties of a cube as it follows:

`L2 = [(shape, cube),(color,gray)]`

`filter_all(L2,L1,[H0|T0])` finds a list of objects with such properties. This is the referenced object. The predicate `get_behind_list(H0,L3)` stores all objects behind the reference object in L3. Next, the objects that are in front of the object(s) in L3 are found and stored in L5. Next, all their properties are stored in L6. It should be noted that "thing_5" represents a different object from "thing_11". Finally, Ids represents the list of objects that satisfy all the criteria, and since this is an existential yes/no question, since the length of this list is greater equal to 1 the predicate `find_ans(Q)` succeeds.

CHAPTER 10

FUTURE WORKS & CONCLUSION

10.1 Future Works

Our heuristics-based algorithms to induce default theories has opened several new avenues for the future work:

1. Handling large datasets using methods similar to QuickFoil (Zeng et al., 2014). In QuickFoil, all the operations of FOIL are performed in a database engine. Such an implementation, along with pruning techniques and query optimization tricks can make FOLD training much faster

2. FOLD learns only function-free answer set programs. We are planning to investigate extending the language bias towards accommodating functions.

3. extending abductive logic programming with abducing rules (as opposed to just ground atoms)

4. LIME-FOLD algorithm in its current form only learns from single table dataset. We are planning to extend LIME-FOLD to support multi-relational features.

5. An alternative approach is to regard the ILP problem as an optimization problem where objective is to maximize the coverage of positive examples and minimize the coverage of negative examples with the least number of clauses possible. *Gradient-descent* and *back-propagation* are two very common solutions to the optimization problems in machine learning. The latter is a recursive algorithm that is used to train neural networks. A key strength of neural networks is that they are robust to noise and mislabeled data, something that most ILP systems suffer from. This has motivated some recent work on combining the ILP and neural networks (Evans and Grefenstette, 2018). However,

the negation-As-Failure (NAF) semantics is absent in this work. We will explore how this differentiable learning can be extended to learn answer set programs.

10.2 Conclusions

In this dissertation, we presented algorithms to induce hypotheses in the form of default theories. To the best of our knowledge, FOLD algorithm presented in our work, is the first heuristics-based ILP algorithm capable of learning non-monotonic logic programs. Extensive experiments suggest that our FOLD algorithm and its extensions, outperform the state-of-the-art ILP systems in terms of the classification evaluation and also the number of induced rules. The most important contributions of this dissertation to the field of Explainable AI is an extension of FOLD algorithm (i.e., LIME-FOLD) to capture the underlying logic of black-box machine learning models also using default theories. Another contribution to Explainable AI is our answer set programming (ASP)-based approach to counter-factual explanation. To the best of our knowledge, this approach is the only solution that respects the logical, physical and temporal constraints during the search for a counter-factual case. Finally, we introduced AQuA, a fully explainable framework based on logic programming to tackle the task of visual question answering. Our framework, that restricts the application of neural networks to the computer vision, is capable of explaining its answers using the resolution proof tree, and unlike any deep-learning-based system, is able to trace all the mistakes made by the system back to the source of the error.

REFERENCES

Aggarwal, C. C. and J. Han (2014). *Frequent Pattern Mining*. Springer Publishing Company, Incorporated.

Agrawal, R. and R. Srikant (1994). Fast algorithms for mining association rules in large databases. In *Proc. of 20th International Conference on Very Large Data Bases*, VLDB '94, CA, USA, pp. 487–499. Morgan Kaufmann Publishers Inc.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge, New York, Melbourne: Cambridge University Press.

Basu, K., F. Shakerin, and G. Gupta (2020). Aqua: Asp-based visual question answering. In E. Komendantskaya and Y. A. Liu (Eds.), *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, Volume 12007 of *Lecture Notes in Computer Science*, pp. 57–72. Springer.

Cao, Q., X. Liang, B. Li, G. Li, and L. Lin (2018). Visual question reasoning on general dependency tree. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7249–7257.

Catlett, J. (1991). Megainduction: A test flight, proceedings of the eighth international workshop (ml91),northwestern university, evanston, illinois,. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*, pp. 596–599. Morgan Kaufmann.

Chen, T. and C. Guestrin (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD*, KDD '16, pp. 785–794.

Corapi, D., A. Russo, and E. Lupu (2012). *Inductive Logic Programming in Answer Set Programming*, pp. 91–97. Berlin, Heidelberg: Springer Berlin Heidelberg.

Cortes, C. and V. Vapnik (1995, September). Support-vector networks. *Mach. Learn. 20*(3), 273–297.

Craig, W. (1957). Linear reasoning: A new formof the herbrand-gentzen theorem. *Journal of Symbolic Logic 22(3)*, 250–268.

Craven, M. W. and J. W. Shavlik (1995). Extracting tree-structured representations of trained networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, Cambridge, MA, USA, pp. 24–30. MIT Press.

Davidson, D. (2001). *Inquiries into truth and interpretation: Philosophical essays*, Volume 2. Oxford University Press.

Diederich, J. (2008). Rule extraction from support vector machines: An introduction. In *Rule extraction from support vector machines*, pp. 3–31. Springer.

Dimopoulos, Y. and A. C. Kakas (1995). Learning non-monotonic logic programs: Learning exceptions. In N. Lavrac and S. Wrobel (Eds.), *Machine Learning: ECML-95, 8th European Conference on Machine Learning, Heraclion, Crete, Greece, April 25-27, 1995, Proceedings*, Volume 912 of *Lecture Notes in Computer Science*, pp. 122–137. Springer.

Evans, R. and E. Grefenstette (2018, January). Learning explanatory rules from noisy data. *J. Artif. Int. Res. 61*(1), 1–64.

Fayyad, U. M. and K. B. Irani (1993a). Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pp. 1022–1029.

Fayyad, U. M. and K. B. Irani (1993b). Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pp. 1022–1029.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Bradford Books.

Fournier-Viger, P., J. Chun-Wei Lin, T. Truong-Chi, and R. Nkambou (2019). *A Survey of High Utility Itemset Mining*, pp. 1–45. Springer International Publishing.

Fournier-Viger, P., J. C. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam (2016). The SPMF open-source data mining library version 2. In *ECML/PKDD (3)*, Volume 9853 of *LNCS*, pp. 36–40. Springer.

Friedman, J. H., B. E. Popescu, et al. (2008). Predictive learning via rule ensembles. *The Annals of Applied Statistics 2*(3), 916–954.

Fung, G., S. Sandilya, and R. B. Rao (2005). Rule extraction from linear support vector machines. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 32–40. ACM.

Gan, W., J. C. Lin, P. Fournier-Viger, H. Chao, T. Hong, and H. Fujita (2018). A survey of incremental high-utility itemset mining. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 8*(2).

Gao, H., J. Mao, J. Zhou, Z. Huang, L. Wang, and W. Xu (2015). Are you talking to a machine? dataset and methods for multilingual image question. In *NIPS'15*, pp. 2296–2304.

Gebser, M., B. Kaufmann, and T. Schaub (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

Gelfond, M. and Y. Kahl (2014). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The ASP Approach*. Cambridge University Press.

Gelfond, M. and V. Lifschitz (1988). The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen (Eds.), *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pp. 1070–1080. MIT Press.

Gunning, D. (2015). Explainable artificial intelligence (xai).

Gupta, G. (2017). A case for query-driven predicate asp. In *ARCADE'17,1st Int. Workshop on Automated Reasoning,Sweden,2017*, pp. 64–68.

Hu, R., J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko (2017). Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 804–813.

Hudson, D. A. and C. D. Manning (2018). Compositional attention networks for machine reasoning. *arXiv preprint arXiv:1803.03067 1*, –.

Huysmans, J., B. Baesens, and J. Vanthienen (2006). Iter: an algorithm for predictive regression rule extraction. In *International Conference on Data Warehousing and Knowledge Discovery*, pp. 270–279. Springer.

Huysmans, J., R. Setiono, B. Baesens, and J. Vanthienen (2008). Minerva: Sequential covering for rule extraction. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 38*(2), 299–309.

Inoue, K. and Y. Kudoh (1997). Learning extended logic programs. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'97, San Francisco, CA, USA, pp. 176–181. Morgan Kaufmann Publishers Inc.

Johnson, J. et al. (2017). Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2989–2998.

Johnson, J., B. Hariharan, L. van der Maaten, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *IEEE CVPR'17*, pp. 2901–2910.

Kramer, S., N. Lavrač, and P. Flach (2000). Relational data mining. In S. Dězeroski (Ed.), *Relational Data Mining*, Chapter Propositionalization Approaches to Relational Data Mining, pp. 262–286. New York, NY, USA: Springer-Verlag New York, Inc.

Krishna, R., Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, et al. (2017). Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision 123*(1), 32–73.

Landwehr, N., K. Kersting, and L. D. Raedt (2005). nFOIL: Integrating naïve bayes and FOIL. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pp. 795–800.

Landwehr, N., A. Passerini, L. D. Raedt, and P. Frasconi (2006). kFOIL: Learning simple relational kernels. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, MA, USA*, pp. 389–394.

Laugel, T., M.-J. Lesot, C. Marsala, X. Renard, and M. Detyniecki (2018). Comparison-based inverse classification for interpretability in machine learning. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations*, Cham, pp. 100–111. Springer International Publishing.

Law, M., A. Russo, and K. Broda (2014). Inductive learning of answer set programs. In *Logics in Artificial Intelligence - 14th European Conference, JELIA*.

Lichman, M. (2013a). UCI,ml repository, http://archive.ics.uci.edu/ml.

Lichman, M. (2013b). UCI,ml repository, `http://archive.ics.uci.edu/ml`.

Looveren, A. V. and J. Klaise (2019). Interpretable counterfactual explanations guided by prototypes. In *CoRR*, Volume abs/1907.02584.

Lundberg, S. M., G. G. Erion, and S.-I. Lee (2018). Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*.

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

Malinowski, M. and M. Fritz (2014). A multi-world approach to question answering about real-world scenes based on uncertain input. In *NIPS'14*, pp. 1682–1690.

Manning, C. D., M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky (2014). The Stanford CoreNLP natural language processing toolkit. In *ACL System Demonstrations*, pp. 55–60.

Marple, K. and G. Gupta (2012). Galliwasp: A goal-directed answer set solver. In E. Albert (Ed.), *Logic-Based Program Synthesis and Transformation, 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers*, Volume 7844 of *Lecture Notes in Computer Science*, pp. 122–136. Springer.

Marple, K., E. Salazar, and G. Gupta (2017). Computing stable models of normal logic programs without grounding. In *arXiv*.

Mascharka, D., P. Tran, R. Soklaski, and A. Majumdar (2018). Transparency by design: Closing the gap between performance and interpretability in visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4942–4950.

McMillan, K. L. (2003). Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi (Eds.), *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, Volume 2725 of *Lecture Notes in Computer Science*, pp. 1–13. Springer.

McMillan, K. L. (2004). An interpolating theorem prover. In K. Jensen and A. Podelski (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, pp. 16–30. Springer Berlin Heidelberg.

Mitchell, T. M. (1980). The need for biases in learning generalizations. In J. W. Shavlik and T. G. Dietterich (Eds.), *Readings in Machine Learning*, pp. 184–191. Morgan Kauffman. Book published in 1990.

Mitchell, T. M. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill.

Muggleton, S. (1991a, February). Inductive logic programming. *New Gen. Comput. 8*(4), –.

Muggleton, S. (1991b). Inductive logic programming. *New Generation Comput. 8*(4), 295–318.

Muggleton, S. (1995a). Inverse entailment and progol. *New Generation Comput. 13*(3&4), 245–286.

Muggleton, S. (1995b, Dec). Inverse entailment and progol. *New Generation Computing 13*(3), 245–286.

Muggleton, S. and W. L. Buntine (1988). Machine invention of first order predicates by inverting resolution. In J. E. Laird (Ed.), *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988*, pp. 339–352. Morgan Kaufmann.

Muggleton, S., H. Lodhi, A. Amini, and M. J. E. Sternberg (2005). Support vector inductive logic programming. In A. Hoffmann, H. Motoda, and T. Scheffer (Eds.), *Discovery Science*, Berlin, Heidelberg. Springer Berlin Heidelberg.

Muggleton, S., L. Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan (2012, January). Ilp turns 20. *Mach. Learn. 86*(1), 3–23.

Muggleton, S. H. and C. H. Bryant (2000). Theory completion using inverse entailment. In J. Cussens and A. Frisch (Eds.), *ILP*, pp. 130–146. Springer Berlin Heidelberg.

Nuñez, H., C. Angulo, and A. Català (2002, 01). Rule extraction from svm. In *Proc. The European Symposium on Artificial Neural Networks*, pp. 107–112.

Núñez, H., C. Angulo, and A. Català (2002). Rule extraction from support vector machines. In *In Proceedings of European Symposium on Artificial Neural Networks*, pp. 107–112.

Perez, E. et al. (2018). Film: Visual reasoning with a general conditioning layer. In *AAAI'18*.

Plotkin, G. D. (1971). A further note on inductive generalization, in machine intelligence, volume 6, pages 101-124.

Quinlan, J. R. (1990a). Learning logical definitions from relations. *Machine Learning 5*, 239–266.

Quinlan, J. R. (1990b). Learning logical definitions from relations. *Machine Learning 5*, 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic 7*(3), 329 – 340. Special Issue: Abduction and Induction in AI.

Redmon, J., S. K. Divvala, R. B. Girshick, and A. Farhadi (2016). You only look once: Unified, real-time object detection. In *CVPR*, pp. 779–788. IEEE Computer Society.

Reiter, R. (1980). A logic for default reasoning. *Artif. Intell. 13*(1-2), 81–132.

Ren, M., R. Kiros, and R. Zemel (2015). Exploring models and data for image question answering. In *NIPS'15*, pp. 2953–2961.

Ribeiro, M. T., S. Singh, and C. Guestrin (2016). "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD 2016*, pp. 1135–1144.

Riguzzi, F. (2016). *ALEPH in SWI-Prolog* , `https://github.com/friguzzi/aleph`. -.

Rümmer, P. (2008). A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Volume 5330 of *LNCS*, pp. 274–289. Springer.

Sakama, C. (2005). Induction from answer sets in nonmonotonic logic programs. *ACM Trans. Comput. Log. 6*(2), 203–231.

Sakama, C. and K. Inoue (2009). Brave induction: a logical framework for learning from incomplete information. *Machine Learning 76*(1), 3–35.

Santor, A. et al. (2017). A simple neural network module for relational reasoning. In *NIPS'17*, pp. 4967–4976.

Seitzer, J. (1997). Stable ilp : Exploring the added expressivity of negation in the background knowledge. In *IJCAI-97 Workshop on Frontiers of ILP*.

Shah, S., A. Mishra, N. Yadati, and P. P. Talukdar (2019). Kvqa: Knowledge-aware visual question answering. In *AAAI*.

Shakerin, F. and G. Gupta (2019). Induction of non-monotonic logic programs to explain boosted tree models using lime. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Volume 33, pp. 3052–3059.

Shakerin, F., E. Salazar, and G. Gupta (2017). A new algorithm to automate inductive learning of default theories. *TPLP 17*(5-6), 1010–1026.

Singleton, P. and F. Dushin (2003). Jpl a java interface to prolog. http://www.swi-prolog.org/packages/jpl/java_api.

Srinivasan, A. (2001). *The Aleph Manual*. -. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

Srinivasan, A., S. Muggleton, and M. Bain (1996). Distinguishing exceptions from noise in non-monotonic learning,in s. muggleton and k. furukawa, editors, second international inductive logic programming workshop (ilp92).

Suarez, J., J. Johnson, and F.-F. Li (2018). Ddrprog: A clevr differentiable dynamic reasoning programmer. In -.

Tseng, V. S., C.-W. Wu, P. Fournier-Viger, and S. Y. Philip (2016). Efficient algorithms for mining top-k high utility itemsets. *IEEE Transactions on Knowledge and data engineering 28*(1), 54–67.

Voigt, P. and A. v. d. Bussche (2017). *The EU General Data Protection Regulation (GDPR): A Practical Guide* (1st ed.). Springer Publishing Company, Incorporated.

Wachter, S., B. Mittelstadt, and C. Russell (2017). Counterfactual explanations without opening the black box: Automated decisions and the gdpr. *Harv. JL & Tech. 31*, 841.

Wielemaker, J., T. Schrijvers, M. Triska, and T. Lager (2012). SWI-Prolog. *Theory and Practice of Logic Programming 12*(1-2), 67–96.

Wu, X., V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg (2007, December). Top 10 algorithms in data mining. *Knowl. Inf. Syst. 14*(1), 1–37.

Yang, Z., X. He, J. Gao, L. Deng, and A. J. Smola (2015). Stacked attention networks for image question answering. *CVPR'16 1*, 21–29.

Yi, K. et al. (2018). Neural-symbolic VQA: Disentangling reasoning from vision and language understanding. In *NIPS'18*, pp. 1031–1042.

Yu, L., E. Park, A. C. Berg, and T. L. Berg (2015). Visual madlibs: Fill in the blank image generation and question answering. In *arXiv*.

Zeng, Q., J. M. Patel, and D. Page (2014, November). Quickfoil: Scalable inductive logic programming. *Proc. VLDB Endow. 8*(3), 197–208.

BIOGRAPHICAL SKETCH

Farhad Shakerin was born on April 10, 1983, in Tehran, Iran. He received his Bachelor of Science in Computer Engineering - Software from Iran University of Science & Technology in October 2006. He started his professional career at Maharan Engineering Corp. in April 2008 as a C++ embedded software engineer. He became an expert in developing safety-critical systems in railway signaling industry. In April 2012, he got promoted to director of the software department at Maharan Engineering Corp., until August 2014 when he decided to join graduate school to further study software verification and formal methods. In the very first semester, he took a course on "Advanced Programming Languages Semantics" under Dr. Gupta and it was then when he first fell in love with Logic Programming and never looked back. He has since been working on symbolic machine learning and explainable AI and has published several papers on the topic.

CURRICULUM VITAE

Farhad Shakerin

March 08, 2020

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: farhad.shakerin@utdallas.edu

Educational History:

B.S., Computer Engineering - Software, Iran University of Science & Technology, 2005
M.S., Computer Science, University of Texas at Dallas, 2016

Employment History:

Software Engineer, Maharan Engineering Corp., April 2008 – August 2014