NNT : 2017SACLX037

Thèse de doctorat de l'Université Paris-Saclay préparée à l'École Polytechnique

École doctorale n°580 : Sciences et Technologies de l'Information et de la Communication (STIC)

Spécialité de doctorat : Informatique

Par

M. Olivier Wang

Adaptive Rules Models: Analytics Learning for Rule-Based Systems

Thèse soutenue et présentée à Palaiseau, le 28 juin 2017

Composition du jury :

Mme. Ioana Manolescu,	Directrice de Recherche LIX - INRIA	Président
M. Adrian Paschke,	Professeur Freie Universität Berlin	Rapporteur
M. Marco Gavanelli,	Professeur Associé Université de Ferrare	Rapporteur
Claudia D'Ambrosio,	Maître de Conférences LIX - INRIA	Examinateur
Mme. Catherine Faron Zucker,	Maître de Conférences Université de Nice Sophia Antipolis	Examinateur
M. Nicolas Changhai Ke,	Architecte Produit BRMS IBM France	Examinateur
M. Leo Liberti,	Directeur de Recherche LIX - INRIA	Directeur de thèse

Abstract

Business Rules (BRs) are a commonly used tool in industry for the automation of repetitive decisions. The emerging problem of adapting existing sets of BRs to an ever-changing environment is the motivation for this thesis. Existing Supervised Machine Learning techniques can be used when the adaptation is done knowing in detail which is the correct decision for each circumstance. However, there is currently no algorithm, theoretical or practical, which can solve this problem when the known information is statistical in nature, as is the case for a bank wishing to control the proportion of loan requests that its automated decision service forwards to human experts. We study the specific learning problem where the aim is to adjust the BRs so that the decisions are close to a given average value.

To do so, we consider sets of Business Rules as programs. After formalizing some definitions and notations in Chapter 2, the BR programming language defined this way is studied in Chapter 3, which proves that no algorithm exists to learn Business Rules with a statistical goal in the general case. We then restrain the scope to two common cases where BRs are limited in some way: the Iteration Bounded case in which no matter the input, the number of rules executed when taking the decision is less than a given bound; and the Linear Iteration Bounded case in which rules are also all written in Linear form. In those two cases, we later produce a learning algorithm based on Mathematical Programming which can solve this problem. We briefly extend this theory and algorithm to other statistical goal

learning problems in Chapter 5, before presenting the experimental results of this thesis in Chapter 6. The latter includes a proof of concept to automate the main part of the learning algorithm which does not consist in solving a Mathematical Programming problem, as well as some experimental evidence of the computational complexity of the algorithm.

Although the algorithms used in Business Rules management systems have been studied and compared, the theoretical study of BR as a programming language has not attracted interest from the research community before now. We dedicate Chapter 3 to this study. We prove the Turing-completeness of this language, which is often assumed but was never proven before, to the best of our knowledge. Of the two proofs we provide, we use the constructive proof to define a canonical equivalence between BR programs and WHILE programs. We use the WHILE form of BR programs to describe a Structural Operational Semantics for BR programs, and showcase the usefulness of this semantics by using it to prove the termination of some BR programs. The last property of BR programs we prove in that chapter has a proof based on the Turing-completeness of BR programs and answers part of the question which motivated this thesis: there is no algorithm which learns Business Rules with a statistical goal in the general case.

The contents of Chapter 4 alternate definitions and algorithms. We describe Iteration Bounded Business Rules (IBBR) programs, which are such that the number of Business Rules executed to take any given decision is bounded by a known value. We exhibit a Mixed-Integer Programming (MIP) problem such that, for IBBR programs, solving it is equivalent to solving the statistical goal learning problem. Similarly, we describe Linear Iteration Bounded Business Rules (LIBBR) programs, which are Iteration Bounded Business Rules programs such that the Business Rules follow a given linear template. The associated algorithm produces a Mixed-Integer Linear Programming (MILP) problem. We evaluate the theoretical complexity of

this algorithm.

We extend the formalism and learning algorithms presented so far to a different class of learning problems in Chapter 5. We exhibit a Mathematical Programming equivalent to LIBBR programs in the case where the known information is over a quantized frequency distribution of the decisions taken, rather than over the average decision taken. In two specific such cases, that of the distribution respecting an upper bound on the frequency of a specific decision and that of the distribution being almost uniform, we also exhibit a MILP problem which is equivalent when the BR program is a LIBBR program.

In Chapter 6, we provide the proof of concept showing how our learning algorithm can be automated to be integrated into a BR management software. The numerical part of our experimental work is in turn entirely dedicated to evaluating the MILP obtained from a LIBBR program using the algorithm described in Chapter 4. The evidence obtained shows that this direct application cannot be scaled up to industrial BR programs. We evaluate the validity of our algorithm by testing the number of randomly generated BR programs that our algorithm can solve. We also evaluate the performance by varying different characteristics of the learning problem and observing the CPU time taken by a standard Mathematical Programming solver to solve the learning problem.

Résumé

Les Règles Métiers (Business Rules en anglais, ou BRs) sont un outil communément utilisé dans l'industrie pour automatiser des prises de décisions répétitives. Le problème de l'adaptation de bases de règles existantes à un environnement en constante évolution est celui qui motive cette thèse. Des techniques existantes d'Apprentissage Automatique Supervisé peuvent être utilisées lorsque cette adaptation se fait en toute connaissance de la décision correcte à prendre en toutes circonstances. En revanche, il n'existe actuellement aucun algorithme, qu'il soit théorique ou pratique, qui puisse résoudre ce problème lorsque l'information connue est de nature statistique, comme c'est le cas pour une banque qui souhaite contrôler la proportion de demandes de prêt que son service de décision automatique fait passer à des experts humains. Nous étudions spécifiquement le problème d'apprentissage qui a pour objectif d'ajuster les BRs de façon que les décisions prises aient une valeur moyenne donnée.

Pour ce faire, nous considérons les bases de Règles Métiers en tant que programmes. Après avoir formalisé quelques définitions et notations dans le Chapitre 2, le langage de programmation BR ainsi défini est étudié dans le Chapitre 4, qui prouve qu'il n'existe pas d'algorithme pour apprendre des Règles Métiers avec un objectif statistique dans le cas général. Nous limitons ensuite le champ d'étude à deux cas communs où les BRs sont limités d'une certaine façon : le cas Borné en Itérations dans lequel, quelles que soient les données d'entrée, le nombre de règles

exécutées en prenant la décision est inférieur à une borne donnée ; et le cas Linéaire Borné en Itérations dans lequel les règles sont de plus écrites sous forme Linéaire. Dans ces deux cas, nous produisons par la suite un algorithme d'apprentissage basé sur la Programmation Mathématique qui peut résoudre ce problème. Nous étendons brièvement cette formalisation et cet algorithme à d'autres problèmes d'apprentissage à objectif statistique dans le Chapitre 5, avant de présenter les résultats expérimentaux de cette thèse dans le Chapitre 6. Ce dernier inclut une preuve de concept pour l'automatisation de la partie principale de l'algorithme d'apprentissage qui n'est pas celle où l'on résout un problème de Programmation Mathématique, ainsi que des indications expérimentales sur la complexité computationelle de l'algorithme.

Bien que les algorithmes utilisés dans les systèmes de gestion des Règles Métiers aient été étudiés et comparés, l'étude théorique des BRs en tant que langage de programmation n'a pas attiré l'intérêt de la communauté scientifique jusqu'à présent. Nous dédions le Chapitre 3 à cette étude. Nous prouvons la Turing-complétude de ce langage, qui est souvent supposée vraie mais n'a jamais été prouvée à ce jour, autant que nous le sachions. Des deux preuves que nous fournissons, nous utilisons la preuve constructive pour définir une équivalence canonique entre les programmes BR et les programmes WHILE. Nous utilisons la forme WHILE des programmes BR pour décrire une Sémantique Opérationnelle Structurelle pour les programmes BR, et mettons en évidence l'utilité de cette sémantique en l'utilisant pour prouver la terminaison de quelques programmes BR. La dernière propriété des programmes BR que nous prouvons dans ce chapitre a une preuve basée sur la Turing-complétude des programmes BR et répond à une partie de la question qui a motivé cette thèse : il n'existe pas d'algorithme qui apprend les Règles Métiers avec un objectif statistique dans le cas général.

Le Chapitre 4 contient une alternance de définitions et d'algorithmes. Nous y décrivons les programmes de Règles Métiers Bornés en Itérations (Iteration Bounded Business Rules en anglais, soit IBBR), qui sont tels que le nombre de Règles Métiers exécutées pour prendre une décision donnée est borné par une valeur connue. Nous exhibons un problème de Programmation Mixte en Nombres Entiers (Mixed-Integer Programming en anglais, ou MIP) tel que le résoudre soit équivalent, pour un programme IBBR, à résoudre le problème d'apprentissage à but statistique. De la même façon, nous décrivons les programmes de Règles Métiers Linéaires Bornés en Itérations (Linear Iteration Bounded Business Rules en anglais, soit LIBBR), qui sont des programmes IBBR tels que les Règles Métiers suivent un modèle linéaire donné. L'algorithme associé produit un problème de Programmation Linéaire Mixte en Nombres Entiers (Mixed-Integer Linear Programming en anglais, ou MILP). Nous évaluons la complexité théorique de cet algorithme.

Nous étendons le formalisme et les algorithmes d'apprentissage présentés jusque là à une classe de problèmes d'apprentissages différente dans le Chapitre 5. Nous exhibons un équivalent en Programmation Mathématique aux programmes LIBBR dans le cas où l'information connue couvre une distribution quantisée des décisions prises, plutôt que la moyenne des décisions prises. Dans deux cas spécifiques, celui où la distribution respecte une borne supérieure sur la fréquence d'une décision particulière et celui où la distribution est presque uniforme, nous exhibons également un problème MILP qui est équivalent lorsque le programme BR est un programme LIBBR.

Dans le Chapitre 6, nous produisons une preuve de concept qui montre comment notre algorithme d'apprentissage peut être automatisé pour être intégré dans un logiciel de gestion de BR. La partie numérique de notre travail expérimental est quant à elle dédiée entièrement à l'évaluation du MILP obtenu à partir d'un programme LIBBR en utilisant l'algorithme décrit dans le Chapitre 4. Les indications

obtenues montrent que cette application directe ne peut pas être mise à l'échelle pour s'appliquer à des programmes BR industriels. Nous évaluons la validité de notre algorithme en testant le nombre de programmes BR générés aléatoirement que notre algorithme peut résoudre. Nous évaluons également la performance en faisant varier différentes caractéristiques du problème d'apprentissage et en observant le temps CPU qu'un solver de Programmation Mathématique standard met à résoudre le problème d'apprentissage.

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Leo Liberti, who has followed, pushed and supported my work for the whole length of my PhD study. His sincere wish to have me produce the best research possible made all of this thesis possible. I could not have had better advisor for my PhD study.

Special thanks also go to my supervisors at IBM. My administrative supervisors M. Patrick Albert, who pushed for IBM to fund this PhD and recruited me, and his successor M. Christian De Sainte Marie, who guided me during my research and the writing of this thesis, were both essential to the success of my thesis.

I am grateful to all the members of my thesis committee. Prof. Adrian Paschke and Prof. Marco Gavanelli did a wonderful work as referees – their comments have led to improvements in the text of this thesis in many places, notably in Chapter 1. Prof. Ioana Manolescu and Prof. Catherine Faron Zucker were wonderful members of the jury with insightful comments and questions. Prof. Claudia D'Ambrosio was all that and more, as she also helped with the Mathematical Programming part of my PhD study. M. Changhai Ke was also my scientific supervisor at IBM, and helped me explore the potential of Business Rules both during my thesis and my defense.

A big thank you also to the rest of the team at the LIX: Sonia, Raouia, Claire, Pierre-Louis, Gustavo, Luca, Ky, Youcef, all of you were great friends and co-workers. Special mention to Pierre-Louis and Sonia for helping with brainstorming sessions and to Gustavo for giving me tips about the LIX servers and the thesis formatting! I cannot forget grouchy Evelyne either, our ever-busy and ever-helpful secretary, she is a marvel.

This thesis would not have been the same without my fellow PhD students at IBM either. Oumaima, Karim, Hamza, Nicolas, Penelope, and Reda, we experienced the wonders of working on a thesis at the IBM lab together, and I would not have it any other way.

Last but not least, I thank all the friends and family who supported me through this thesis: my friends Eric and France (and their cat Styx), my brother and sister, my parents, and my wonderful girlfriend Isabelle. You all bring joy to my life, and I love you.

Contents

Glossary

BR	Business Rules
BRMS	Business Rules Management System
IBBR	Iteration Bounded Business Rule(s)
iff	if and only if
LIBBR	Linear Iteration Bounded Business Rules(s)
MIP	Mixed-Integer Programming
MILP	Mixed-Integer Linear Programming
MINLP	Mixed-Integer Nonlinear Programming
ML	Machine Learning
MP	Mathematical Programming
ODM	Operational Decision Management
PAC	Probably Approximatively Correct
POC	Proof of Concept
PRF	Pseudorandom Function
SOS	Structural Operational Semantics
TM	Turing Machine
UTM	Universal Turing Machine

Chapter 1

Introduction

This thesis is concerned with tuning parameters in Business Rules programs so that a given statistical measure over all the decisions taken by a given Business Rules (BR) program satisfies a given objective. Business rules are used in industry today as a functional way to formally encode corporate policies and regulations. For example, the decision of whether to grant a loan request, refuse it, or mark it for further review by a human expert is often made by a BR program. Of course, the decision should be made on a per-customer basis, and yet there may be reasons on a corporate scale to wish for a certain proportion of all loans to be reviewed by human experts, while the rest can be treated automatically. We investigate how the rules in a BR program can be tuned automatically, e.g. to adapt to changes in the reviewing policy or to evolutions in the market.

In this introduction, we first explain the reasons motivating this thesis, then we outline the main contributions of this thesis, including publications. Finally, we give an overview of the methodology followed in this manuscript.

1.1 Motivation

This PhD was sponsored by IBM France Lab, the R&D unit in IBM which is responsible for the development and support of IBM Operational Decision Management (ODM), one of the most popular Business Rules Management System (BRMS) on the market [57]. A very common request clients make to the ODM support team is help in creating a BR program which outputs the correct decisions for the example inputs they already have, while keeping the structure of the rules modeled by their business experts. The defining characteristics of a correct decision differ from client to client.

The main motivation for this thesis was the realization that many of IBM's clients have a statistical definition of what makes a decision model 'correct'. No tool or theory exists to parametrize an existing BR program with the objective of "thirty percent of decisions must be reviewed manually over the example cases". Studying the possibility and practicality of creating such a tool is the main objective of this thesis from the point of view of IBM.

Outside the specific environment of BRMS vendors such as IBM, other situations also call for this type of statistical parametrization. In many situations, the existence of an algorithm for learning a program with a statistical goal would be very practical, as is the case in the medical field when trying to match ADN sequences and genetic predisposition to illnesses. Learning statistical goals is a problem with many important industrial applications, yet little formal research. In this thesis, we explore a possible solution to this problem when applied to BR programs.

1.2 The contributions

During this thesis, we have proved that no algorithm can ever guarantee an exact conformance to given statistics over all possible cases: the task of learning the behavior of BR programs in general is formally impossible. This general result does not prevent some subset of BR programs to be learnable, including some that are of particular interest to us: BR programs that have a bounded execution time and BR programs that have only linear rules. This result stems from the Turing-completeness of the BR programming language, which is widely known yet had avoided published formal proof for a long time.

From the Turing-completeness of the BR programming language, we have described a WHILE form of BR programs which we used to define a Structural Operational Semantics (SOS) for BR programs. We have shown that this SOS can be used to prove properties of BR programs such as termination over a range of inputs.

The most relevant result for out initial research question is the creation of a learning algorithm for the statistical goal learning problem applied to Iteration-Bounded BR (IBBR) programs. This algorithm based on Mathematical Programming can be used to transform a statistical goal learning problem into Mixed-Integer Linear Problems (MILPs) in the case of Linear IBBR (LIBBR) programs. We have applied this algorithm and evaluated its algorithmic complexity using empirical solving times obtained with a standard MILP solver. We have also extended the original algorithm to different statistical goals that the one it was originally created for, and used it to solve two additional types of statistical goal learning problems.

The theoretical impossibility result was published with a formal proof of the Turing-completeness in [140]. The description and testing of the original learning algorithm was published in [138]. The general statistical goal learning problem and the extension of the algorithm to different statistical goals was published in [139].

1.3 The approach

In Chapter 2, we define the notions and notations which will be used in the rest of this thesis, and present the existing research relevant to our work in the domains of Business Rules, Machine Learning and Mathematical Programming. We notably present a formal definition of Business Rules in Sec. 2.1, then a formal definition of the statistical goal learning problem in Sec. 2.2.

We then seek to answer the question of the existence of an algorithm which can solve the learning problem of interest in this thesis, that of learning parameters of a BR program when given a statistical goal, in the general case. Chapter 3 is dedicated to studying this question. We give a negative answer in Sec. 3.1 by proving the Turing-compleness of BR programs, and derive a useful operational semantics for BR programs from our proof. This operational semantics can in particular be used to prove termination of a BR program over an interval of values, by encoding the BR semantics into an inference engine such as Prolog [99], as demonstrated in Subsec. 3.2.3. We then further generalize the nonexistence of a learning algorithm in Sec. 3.3 by proving not only PAC-unlearnability, but an even stronger unlearnability property.

In the rest of the thesis, we study a MP-based algorithm for solving the statistical learning problem when restricted to some subclasses of BR programs. We introduce Iteration Bounded BR programs (IBBR) and Linear Iteration Bounded BR programs (LIBBR) in Chapter 4. In each case, we provide an equivalent MP problem to the learning problem formalized in Sec. 2.2. We show that this problem can be solved for a LIBBR program by a MILP solver in Sec. 4.4.

We briefly touch on different forms of statistical learning problems in Chapter 5. In particular, we provide MP formulations for two different problems: the problem where the proportion of outputs taking a particular value (e.g. o_1) is bounded, i.e. $\text{card}(\{q \in Q \mid P(p,q) = o_1\})/\text{card}(Q) \leq g$, and the problem where the outputs must respect an almost uniform distribution. Those formulations are also MILP problems when the BR programs are LIBBR programs.

We also cover some practical tests to evaluate the practicality of our algorithm in Chapter 6. We include a practical way of creating the MILP problem for a BR program written with IBM Operational Decision Management (ODM), as well as a few tests on the different forms of statistical learning problems.

We conclude this thesis with a discussion of our contributions and of the research perspectives this work opens in Chapter 7.

Chapter 2

Definitions and state of the art

In this Chapter we introduce the three key concepts which we make use of in this thesis: Business Rules (BRs), Machine Learning (ML), and Mathematical Programming (MP). In the first section, we introduce the notion of Business Rules used in industry and research, then formally define BR programs. In the second section, we formalize our learning problem and relate our problem to conventional Machine Learning theory. The third section introduces Mathematical Programming definitions and conventions, as MP is the main component of the learning algorithm introduced in this thesis.

2.1 Business Rules (BRs)

BRs are a rules-based programming language in which writing a program consists of defining rules such as "**If** the person requesting this loan has negative balance on a bank account, **Then** refuse this loan request and watch that account". BRs are very popular in corporate environments as a way of automating decision management.

In this section, we first informally explore the uses of BRs in industry and research, and how they relate to this thesis. We then formally define BR programs and their interpreters, and we justify the simplifications used in the rest of the thesis compared to the BR execution algorithms used in commercial products.

2.1.1 BRs in industry and research

Any organization, from the smallest start-up to the largest governmental agency, must constantly make decisions of all kinds, from the most convoluted budget assignment to the simplest YES/NO binary choice. Automated decision management has existed for years as a way to save time and rationalize some of the most repetitive decisions an organization must take. This mostly applies to large companies which have well-defined business policies to be applied in circumstances where either traceability or compliance is needed.

Used for automated decision management, Business Rules are a popular way of modeling operational decisions, such as generating quotes for insurance companies, transaction processing for financial companies, or price computation in retail. Business Rules Management Systems (BRMS) such as IBM Operational Decision Management (ODM), previously known as ILOG JRules, are used by corporations of all sizes to let business experts automate their expertise while still using an easily understandable approximation of natural language [91]. BRMS are commercialized in multiple forms, which also include FICO Decision Management Suite [42], TIBCO Business Process Management Suite [123], Pega Robotic Process Automation [96], RedHat JBoss BRMS [61], OpenRules [93], and ServiceNow [112], and others. The Java Rules engine JESS [62] also sees some industrial use. One of the most popular characteristics of BRMS is their ease of use: business processes are defined by business analysts who do not often have a Computer Science background, which justifies the choice of BRs – as a “programming language for non-programmers” – to encode the decisions which are part of those processes.

This ease of use stems from the fact that BRs have a streamlined syntax which makes writing BR programs as elegant as declarative programming languages. This characteristic is emphasized by using pseudo-natural language to bridge the gap between the machine definition of objects (both classes and methods) and their business definition. Every sector of business can find uses for BRMS, from banks (e.g. for loan validation) to car companies (e.g. for car models and options pricing) to insurance companies (e.g. for fraud detection).

The primary strength of BRs compared to other decision management systems lies in its agility. As business environments change, business policies must inevitably adapt. It has become increasingly frequent due to globalization, which accelerates regulation and market changes. Organizations have to conform with more and more regulations, market demand evolves faster and faster, and competition reacts and must be reacted to with increasingly short delays. BRMS allow business experts with no advanced computer knowledge to write and modify BRs, after the initial intervention of the I.T. department to devise semi-natural, domain-specific rule languages. Furthermore, BRMS include practical lifecycle management of BR programs such as modifying BRs which are already deployed to production, along with the usual versioning and collaborative working features.

The initial idea which later led to BRs appeared under the name of Production Rules in the 1970s as a knowledge representation system [37] originating as a psychological model of human reasoning behavior [85, 86]. In the 1980s, the Artificial Intelligence research community implemented the knowledge representation system used in Production Rules as defined by the modeling community into Expert Systems. They were first theorized [29, 74] then implemented in expert systems such as R1 [143], MYCIN [21, 83], EMYCIN [21, 40], or OPS5 [94, 95]. OPS5 in particular was based on the Rete algorithm [43], which is still in use in many BRMS today. The commercial successors of OPS5 included CLIPS [30], Jess [62], Nexpert [87], and ILOG Rules [59], which evolved into the BR engine currently used by ODM.

As the popularity of Expert Systems faded, the utility of Production Rules as a business decision modeling tool was quickly apparent to the business community, where they gained the name of Business Rules. Although Ronald Ross wrote the very first book about business rules in 1994 [107] and claims to be "the father of business rules" [108], a number of other authors contributed to the definition and expanded use of business rules as an approach to representing business knowledge [53, 54, 18], going as far back as Knolmayer in 1993 [68]. Methodologies have been defined, in particular the Business Rules Manifesto by the Business Rules Group [19].

Expert Systems were progressively replaced by the current form of BRMS, as the decline in popularity of Expert Systems led to the use of standardized BR engines as direct implementations of BRs as defined by the business modeling community: where once Rules engines aimed to intelligently produce a handful of strategic or expert decisions, they now automate thousands of operational and routine decisions.

The defining characteristics of BRMS which any further improvement must take into account and respect are thus automation – the ability to apply one specified action to any number of relevant examples – and accessibility – an ease of use which can allow a Business Analyst to use the interface without issue. The goal of this thesis is to investigate the use of Machine Learning (ML) to modify BRs in an automated fashion. This would allow BRMS to combine their natural advantage – the ease of programming which makes them understandable by business experts – with their ML competitors' best advantage – the ability to take advantage of the vast amounts of data naturally produced by businesses in the age of the Internet.

Rules related programming languages in computer science today can broadly be separated in two categories, which must not be confused [136]:

- Inference systems are closest to logic programming languages, they use Logic Rules and allow for both forward and backward chaining to deduce facts or prove theorems given rules and facts. Prolog [99] is an example of rule-based inference system. Datalog [36] can also be used as a query language for deductive databases. Logic Rules are also used in constraint programming, e.g. in the form of Constraint Handling Rules [27].

- Production systems are closest to standard imperative programming languages, they use Production Rules which can dynamically affect the value of variables at execution time. In query languages such as Datalog¬¬ [1], which use inference engines, this means allowing for negations in heads of rules, interpreted as deletions of facts. In expert systems such as OPS5 [94], which use production engines, this means using *assignment* operations. In terms of logic programming, the presence of assignments means that inference in production systems can be non-monotonic.

The Business Rules we study are part of the second family, they have side effects which in turn affect the conditions of other rules, thus making logical inference impractical (in particular backwards chaining).

Inference systems have been explored in many different ways. Multiple inference languages exist [1] with different reasoning systems, which allow backward chaining, forward chaining, and negation as failure to different degrees. While our focus is not on Logic Rules, it must be noted that many applications of Rules systems use Logic Rules rather than Production Rules – or rather, they fall into the overlap which consists of two functionally equivalent classes of Rule systems: Business Rules where actions do not modify existing variable values; and Logic Rules which only consider forward chaining, with neither backward chaining nor negation as failure. One such example is the research in using a rule-based representation system to organize and execute legal reasoning [71, 70].

Recent research in Production Rules systems has been mostly limited to their applications as a knowledge representation method, such as in [110]. An important part of that focus is dedicated to the combination of Production Rules and Ontologies, as two complementary and synergistic knowledge representations methods: Production Rules focus on the description and execution of business logic while Ontologies focus on the description and structure of domain knowledge [92, 82, 8]. This has led to the SWRL standard proposal [122], which combines the OWL ontology format with the RuleML rules format. Other areas of research in Production Rules systems include handling uncertainty [2] and automated explanation.

We choose to focus on BRs as a commonly used way of encoding business knownledge and automating decisions [46, 58] of interest to IBM. While Production Rules can be represented by declarative logic programs [105, 119], such a transformation does not bring us closer to a learning algorithm for our problem: existing Logic Rules learning algorithms aim to learn relationships, in the fashion of association rule learning (see Sec. 2.2). As we are trying to refine existing BR programs so as to satisfy a statistical goal, we wish to learn parameters in Production Rules rather than to learn Rules themselves, which makes relationship learning irrelevant to our research.

To avoid confusion, we hereafter call *Business Rules* (BRs) the constituents of a computer program meant to be executed by a BRMS or similar BR execution engine, while what is called a business rule by the Business Rules Group is referred to explicitly as a *business modeling rule* when necessary. In the example from earlier, the business modeling rule

> "Loan requests made while the borrower has a negative balance on one of his bank accounts are to be refused, and such accounts must then be watched"

might be decided by a business analyst. Such rules can be written in natural language using an if-then template:

If the person requesting this loan has
  negative balance on a bank account,
Then refuse this loan request
  and watch that account (2.1)

In this example, the phrase "the person requesting this loan has negative balance on a bank account" is the condition, and the phrase "refuse this loan" is the action associated with this business modeling rule. A set of such rules can be written as a BR program given an appropriate representation of the business objects referred to in business modeling rules, e.g. as a well-defined set of objects and classes, called a Business Object Model in ODM.

2.1.2 A formal definition of BRs

The origin of BRs as a knowledge representation system and the later focus on implementations of BRs as business tools have lead to a lack of studies on BRs as a programming language. That is the starting point we choose to use for the contributions in this thesis. A BR is an if-then statement in which if is followed by a boolean expression, called the condition, and then is followed by a sequence of assignment instructions, called the *action*[1]. This action can include non monotonic

[1] In practice, actions can include all sorts of instructions. However, we disregard side-effects such as `print instructions, as our only interest lies in the evolution`

effects, unlike the *consequence* clause of Logic Rules.

The syntax of BRs is straightforward, as it is aimed at non-programmers. In particular, it avoids explicit occurences of loops and function calls, which non-programmers may find difficult to understand. BRs eliminates the need for explicit loops – by making them implicit in the interpreter – and can fulfill the role of function calls (code factorization) by using what we call in this thesis *typed meta-variables.*

Most BRMS use typed meta-variables to simplify their users' task. We explain how those work, then show that the expressive power of such a BR programming language is the same as one that does not use meta-variables at all.

In the rest of this thesis, we often use self-explanatory pseudo-code to describe programming language behavior. The use of the keywords if ... then .. end if, while ... do ... end while, True and False as well as the function *type*() and the symbol $\leftarrow$ do not refer to a specific implementation of computer instructions, but rather to familiar concepts of computer science, namely if-then statements, while loops, Boolean values, variable type evaluation and affectation of a value to a variable. We also use basic arithmetic and boolean syntax.

The existence of typed meta-variables in BRMS is intended to simplify the translation of the production rules defined by a business analyst into BRs used by the software. Let us continue with our example of a bank using BRs to decide whether to automatically accept or reject a loan request. One of the production rules the program must implement might be the one in Eq. 2.1. Without using meta-variables, the BR program must use as many BRs as there are account types, with many rules such as:

```
if thisRequest.borrower.has_saving_account()
        $\wedge$ thisRequest.borrower.saving.balance $< 0$ then
```

of the Working Memory.

```
    thisRequest.accepted ← False
    thisRequest.borrower.saving.watched ← True
end if
```

and

```
if thisRequest.borrower.has_checking_account()
        ∧ thisRequest.borrower.checking.balance < 0 then
    thisRequest.accepted ← False
    thisRequest.borrower.checking.watched ← True
end if
```

However, using meta-variables we can condense these BRs into the definition of the ‘account’ variable type and a single BR with a meta-variable:

```
  # The header of the BR program contains all type
  # declarations, including this new one:
$type(\alpha)$ ← Account
  # The set of Business Rules is now simplified:
if thisRequest.borrower = $\alpha$.owner() ∧ $\alpha$.balance < 0 then
    thisRequest.accepted ← False
    $\alpha$.watched← True
end if
```

When the appropriate BR engine reads this rule, it finds the meta-variable ‘α’ and matches it to each object with the appropriate typing, in this case each account in the input database, identified by having the object type ‘Account’. By contrast, the original items in the database such as ‘thisRequest.borrower’ and ‘thisRequest.accepted’ are input variables with appropriate typing in the database, e.g. ‘Client’ and ‘Boolean’ respectively. The variable ‘thisRequest’ can be seen as a variable with the appropriate typing ‘LoanRequest’, or as the vector of input variables x to make a parallel with the formal definitions introduced below.

In many of the BRMS cited in Subsec. 2.1.1, BRs are written using those meta-variables as well as input variables. We will keep explicitly calling the former *meta-variables*, while shortening the latter to simply *variables*. Both are typed in those BR languages: every meta-variable and variable has a type, declared at the beginning of the BR program. A BR program consists in a set of type, variable and meta-variable definitions, including type declarations, and a set of rules (called in ODM the BOM and the Ruleset respectively). A type declaration consists of the assignment of a type to a variable ($type(\mathsf{var}) \leftarrow \mathsf{type}$), where var can be either a variable or a meta-variable. In our example, the type of 'accepted' is Boolean, and both the variable 'thisRequest.borrower.saving' and the meta-variable 'α' are of type 'Account'.

Definition 2.1 (General form of BRs)**.** Given $\alpha \in \Lambda$ the vector of typed meta-variables and $x \in X$ the vector of typed variables, with X being the set of possible values of x, the general form of a Business Rule is written:

```
if $T(\alpha, x)$ then
    $\alpha' \underset{\text{meta}}{\leftarrow} A(\alpha, x)$
    $x' \leftarrow B(\alpha, x)$
end if
```

where T is a boolean function called the *condition* of the BR; α' (resp. x') represents the value of the variables assigned to α (resp. the value of the variables x) after the execution of the BR; and the couple (A, B) of functions from $\Lambda \times X$ to X describes the *action* of the BR, the action itself being the assignment instructions. By extension, we often simply call (A, B) the action of the BR as well.

The above definition is a proposed format for consistent description of BR programs which is applicable to all BR languages, but is particularly appropriate for adapting BR programs written using the ODM software developped by IBM. The function A corresponds to assignments of new values to variables (components of

x), such as assigning False to ‘accepted’ in our example. The function B corresponds to assignments of values to meta-variables. More precisely, it corresponds to assignment of new values to the variables which were matched to the meta-variables. In our example, we watch the account matched to ‘α’. Many BRMS allow for the use of an else clause in BRs, with the resulting rule having the form:

```
**if** $T(\alpha, x)$ **then**
    $\alpha' \underset{\text{meta}}{\leftarrow} A(\alpha, x)$
    $x' \leftarrow B(\alpha, x)\text{x}$
**else**
    $\alpha' \underset{\text{meta}}{\leftarrow} C(\alpha, x)$
    $x' \leftarrow D(\alpha, x)\text{x}$
**end if**
```

where C and D are also functions from $\Lambda \times X$ to X. This is naturally equivalent to having the two BRs:

```
**if** $T(\alpha, x)$ **then**
    $\alpha' \underset{\text{meta}}{\leftarrow} A(\alpha, x)$
    $x' \leftarrow B(\alpha, x)\text{x}$
**end if**
**if** $\neg T(\alpha, x)$ **then**
    $\alpha' \underset{\text{meta}}{\leftarrow} C(\alpha, x)$
    $x' \leftarrow D(\alpha, x)\text{x}$
**end if**
```

Consequently, we assume in this thesis that all BRs have the form described in Def. 2.1.

Definition 2.2 (General form of BR programs)**.** The general form a BR program is:

# Type definitions

# Input Variables declaration

# Meta-variables declaration

# Type declarations

# Business Rules

At least one of the input variables is selected so that its final value is the *output* of the BR program.

In our example, this corresponds to a BR program being written with the following parts:

# 'LoanRequest', 'Client' and 'Account' are defined to be types, with their respective fields and their type, e.g. 'Account' has fields 'balance' (an integer) and 'watched' (a boolean)

# 'thisRequest' is defined to be the name of the input variable

α is defined to be a meta-variable, potentially among others

# 'thisRequest' is declared to be a 'LoanRequest', thus the vector of input variables includes 'thisRequest.borrower.saving.balance', among others

α is declared to be an 'Account', other meta-variables are typed as well

# The Business Rules

We now give an example of a BR program with two rules. It uses the same variables 'thisRequest.borrower' and 'thisRequest.accepted' as above, as well as the two meta-variables α and β, both of type 'Account':

```
if thisRequest.accepted = False
        ∧ thisRequest.borrower = $\alpha$.owner() ∧ $\alpha$.balance < 0
        ∧ thisRequest.borrower = $\beta$.owner() ∧ $\beta$.balance ≥ 10,000 then
    thisRequest.accepted = True
    $\alpha$.watched ← True
end if
```

```
**if** thisRequest.borrower = $\alpha$.owner() $\wedge$ $\alpha$.balance $< 0 \wedge \alpha$.watched $\neq$ True **then**
    thisRequest.accepted $\leftarrow$ False
    $\alpha$.watched$\leftarrow$ True
**end if**
```

When the initial value of 'thisRequest.accepted' is always True, and no account is initially watched, this BR program encodes the following business modeling rules:

If	the person requesting this loan has
	negative balance on a bank account
	and balance exceeding \$10,000 on another,
Then	accept this loan request
	and watch that account
If	the person requesting this loan has
	negative balance on a bank account
	and no acount with balance over \$10,000,
Then	refuse this loan request
	and watch that account
If	the person requesting this loan has
	no account with negative balance,
Then	accept this loan request

When executing a BR program, meta-variables are stored in a different symbol table than the variables. The meta-variable table matches variables appearing in a BR to variable names, and the variable table matches the latter to stored values. The matching from meta-variables to variables is typed: each meta-variable can only match to a variable of the same type.

In our example, suppose we treat the loan request from John, a customer with two accounts. We call 'thisRequest.borrower.checking' his checking account, with

$-10, and we call 'thisRequest.borrower.saving' his saving account, with $15,000. The BR program's Working Memory initially contains:

Variable	thisRequest .borrower	thisRequest .borrower .checking	thisRequest .borrower .saving	thisRequest .accepted
Matched Value	John's customer information	{balance=-10; watched=False}	{balance=5000; watched=False}	True

When evaluating the first BR's condition, it contains this symbol table and an additional table for meta-variables. The latter first contains:

Meta-Variable	α	β
Matched Variables	thisRequest.borrower .checking	thisRequest.borrower .saving

which lets the condition evaluate to True, and then contains:

Meta-Variable	α	β
Matched Variables	thisRequest.borrower .saving	thisRequest.borrower .checking

which lets the condition evaluate to False. After checking the conditions of the second rule as well, the rule which is executed is found to be the first one with $\alpha =$ thisRequest.borrower.checking and $\beta =$ thisRequest. borrower.saving. At the end of the first rule's execution, the two symbol tables are thus:

Variable	thisRequest .borrower	thisRequest .borrower .checking	thisRequest .borrower .saving	thisRequest .accepted
Matched Value	John's customer information	{balance=-10; watched=True}	{balance=5000; watched=False}	True

Meta-Variable	α	β
Matched Variables	thisRequest.borrower .checking	thisRequest.borrower .saving

John's loan will be accepted, but his checking account will be watched by the bank.

In the rest of this thesis, we assume the header of a BR program to be implicitly complete and coherent: all types, variables and meta-variables are defined; and all variables and meta-variables are typed. Consequently, we often refer to a BR program as a set a BRs, and vice-versa.

A rules engine is an interpreter for a set of rules. Abstract interpretation for a BR program is in two stages: first, the BR program is turned into a set of *elementary rules*, where the meta-variables are replaced by corresponding variables; then, elementary rules are run by an execution algorithm, which includes a conflict resolution strategy to decide the order of rule execution. In practice, those two stages are merged for the sake of computational efficiency: not all elementary rules are generated, instead the rules are instantiated at execution time using the Rete algorithm [43][2]. We choose to describe the theoretical interpreter for clarity, as we only focus on expressive power for the moment.

To create the elementary rules derived from each rule, α is replaced by every type-feasible reordering of the x variable vector. For $x \in X \subseteq \mathbb{R}^d$, i.e. a BR program with $d \in \mathbb{N}$ variables, the explicit set of elementary rules compiled from one rule of the form in Def. 2.1 is the type-feasible part of the following code fragments, using $(\sigma_j \mid j \in \{1, ..., d!\})$ the permutations of $\{1, ..., d\}$:

```
**if** $T((x_{\sigma_1(1)}, ..., x_{\sigma_1(d)}), (x_1, ..., x_d))$ **then**
    $(x_{\sigma_1(1)}, ..., x_{\sigma_1(d)}) \leftarrow A((x_{\sigma_1(1)}, ..., x_{\sigma_1(d)}), (x_1, ..., x_d))$
    $(x_1, ..., x_d) \leftarrow B((x_{\sigma_1(1)}, ..., x_{\sigma_1(d)}), (x_1, ..., x_d))$
```

[2] Readers who have experience with the Rete algorithm will identify the process of matching meta-variables to variables as being part of passing through the Alpha network, while instantiation of BRs happens in terminal nodes of the Beta network.

```
end if
...
if $T((x_{\sigma_d(1)}, ..., x_{\sigma_d(d)}), (x_1, ..., x_d))$ then
    $(x_{\sigma_d(1)}, ..., x_{\sigma_d(d)}) \leftarrow A((x_{\sigma_d(1)}, ..., x_{\sigma_d(d)}), (x_1, ..., x_d))$
    $(x_1, ..., x_d) \leftarrow B((x_{\sigma_d(1)}, ..., x_{\sigma_d(d)}), (x_1, ..., x_d))$
end if
```

The size of this set varies. The typing of the variables and meta-variables matters, and depending on T, A and B, some of these if-then statements might also be computationally equivalent. The number of elementary rules compiled from a given rule can be 0 for an invalid rule ($\forall \alpha, x, T(\alpha, x) = \mathsf{False}$), 1 for a static rule ($T(\alpha, x)$ and $B(\alpha, x)$ do not vary with α, $A(\alpha, x) = \alpha$), and up to $d!$ for some rules if every variable has the same type.

In our example BR program, the set of elementary rules created with this method would contain four rules:

```
if thisRequest.accepted = False ∧ thisRequest.borrower.checking.balance < 0
        ∧ thisRequest.borrower.saving.balance ≥ 10,000 then
    thisRequest.accepted = True
    thisRequest.borrower.checking.watched← True
end if
if thisRequest.accepted = False ∧ thisRequest.borrower.saving.balance < 0
        ∧ thisRequest.borrower.checking.balance ≥ 10,000 then
    thisRequest.accepted = True
    thisRequest.borrower.saving.watched← True
end if
if thisRequest.borrower.checking.balance < 0
        ∧ thisRequest.borrower.checking.watched ≠ True then
    thisRequest.accepted ← False
    thisRequest.borrower.checking.watched$\leftarrow$ True
end if
if thisRequest.borrower.saving.balance $< 0$
        $\wedge$ thisRequest.borrower.saving.watched $\neq$ True then
    thisRequest.accepted $\leftarrow$ False
    thisRequest.borrower.saving.watched$\leftarrow$ True
end if
```

Definition 2.3 (Simplified BR)**.** Given x the vector of variables, the form of an elementary Business Rule (BR) is written:

```
if $T(x)$ then
    $x \leftarrow A(x)$
end if
```

The simplified BR form of BR programs is simply obtained by taking the set of elementary rules obtained from the original BRs in a BR program as a new set of rules, which are more numerous and often less meaningful to a business manager, yet are functionally equivalent.

In the rest of this thesis, we will use the fact that a BR program can be written without loss of generality as a set of elementary BRs, i.e. as a set of rules of the type:

```
if $T(x)$ then
    $x \leftarrow A(x)$
end if
```

The second part of a BR program interpreter is the execution algorithm. We only consider execution algorithms which include a main loop. The standard pattern followed by an execution algorithm with a main loop is conceptually simple:

1. Select as *executable rules* all the instances of elementary rules for which the

condition is True, using the current values of the variables

2. Check a preset Halting condition and stop if it is True

3. Use a preset conflict resolution strategy to select a single rule instance from the set of executable rules selected in step 1.

4. Execute the *action* of the selected executable rule

5. Restart from step 1.

In the rest of this thesis as well as in all commercial BRMS, the default Halting condition in step 2 is True if and only if (iff) there is no executable rule identified in step 1, i.e. if all *condition*s are False. In some BRMS, the Halting condition can also be True if alternate conditions are fulfilled. However, such alternate Halting conditions can always be replaced by the default one, by embedding the Halting test in the rules' conditions. Let the Halting condition be $H(x)$, we simply add an extra condition to each BR. Given a BR with condition $T(x)$, we replace it by $\neg H(x) \wedge T(x)$. We can then replace $H(x)$ by the default Halting condition. As a consequence, we describe no such alternate conditions in this thesis.

The most basic interpreter $\mathscr{I}_0$ for a BR program follows this pattern. The conflict resolution strategy it uses in its execution algorithm consists in a preset total order over all possible elementary rules, which lets the interpreter select the greatest elementary rule for that order relation. The order of elementary rules used is defined by the lexicographic order derived from a predefined order on the rules in the rule set to execute and a predefined order on input variables. This is illustrated in Fig. 2.1. In this thesis, we only consider the basic interpreter, as we can use it to simulate any more complicated ones.

The most common conflict resolution strategies combine one or more of the following three elements [137]:

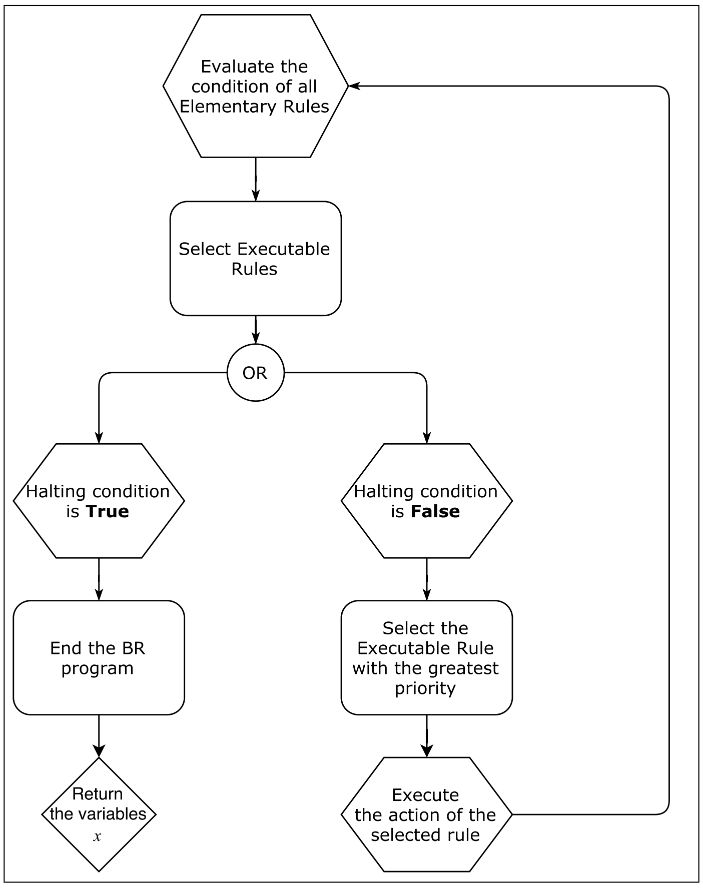

Evaluate the condition of all Elementary Rules
Select Executable Rules
OR
Halting condition is **True**
Halting condition is **False**
End the BR program
Select the Executable Rule with the greatest priority
Return the variables x
Execute the action of the selected rule

Figure 2.1: Control flow corresponding to the basic interpreter

- *Refraction* which prevents an elementary rule from being selected by the conflict resolution algorithm after it is executed unless its *condition* clause has been reset in the meantime.
- *Priority* which is a partial order on rules, leading of course to a partial order on elementary rules.
- *Recency* which orders elementary rules in decreasing order of continued validity duration (when elementary rules are created at run time, it is often expressed as increasing order of elementary rule creation time).

These elements can be simulated by the basic interpreter by adding more types, variables or rules, in broad strokes:

- *Refraction* is simulated with the use of an additional boolean variable (e.g. *needsReset*) per rule per variable permutation (i.e. per elementary rule), an additional test clause and assignment action in each rule, and an additional rule per rule r (**if** $needsReset_{x,r} = \mathsf{True} \wedge T_r(\alpha, x) = \mathsf{False}$ **then** $needsReset_{x,r} \leftarrow \mathsf{False}$).

- *Priority* results in one additional integer variable p, an additional test clause in each rule ($p = \pi_r$), an additional action clause in each rule ($p \leftarrow p_{\max}$), and two additional rules that come dead last in the predefined order on rules (**if** $p > 0$ **then** $p \leftarrow p - 1$; **if** $p = 0$ **then** Stop). The second of those additional rules can in turn be transformed into an additional test clause in each rule, as mentioned earlier.

- *Recency* is the most complicated. A possible simulation could involve an additional integer variable *validityStart* per elementary rule, an additional integer variable *timer*, an additional action clause in each rule ($timer \leftarrow timer + 1$) and a similar setup to the one suggested for *priority*, using an integer variable $priorValidity$ which would this time start at 0 and end at $timer$.

The Business Rules transformed and created by those modifications, with any predefined order which puts the rules created to increment the p and $priorValidity$ counters at the very bottom, simulates an interpreter with those conflict resolution elements in $\mathscr{I}_0$. We call $\mathscr{I}_S$ the standard BR interpreter which uses the aforementioned execution loop as well as those three conflict resolution strategy elements. It is illustrated in Fig. 2.2.

Some degenerate execution algorithms do not include a main loop, as many users of BRMS do not require the recursive ability of the standard interpreter.

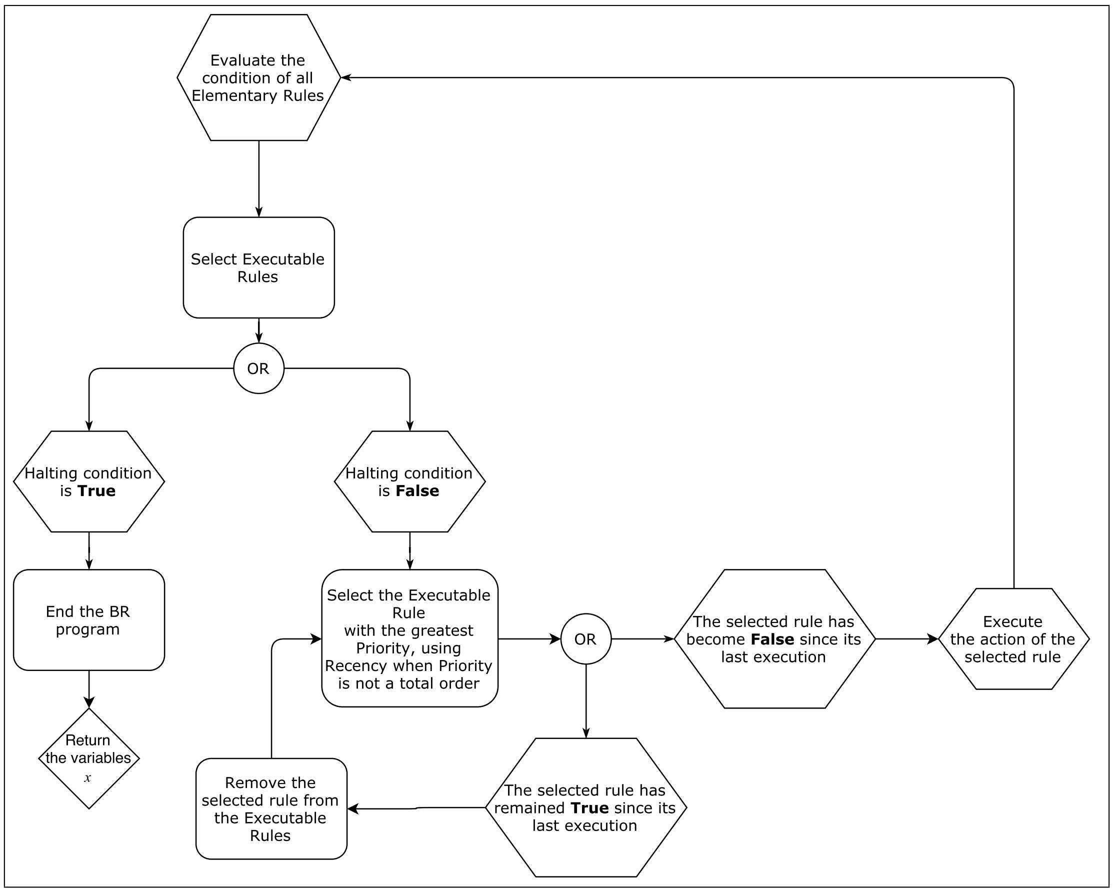

Evaluate the condition of all Elementary Rules
Select Executable Rules
OR
Halting condition is **True**
Halting condition is **False**
End the BR program
Return the variables x
Select the Executable Rule with the greatest Priority, using Recency when Priority is not a total order
OR
The selected rule has become **False** since its last execution
Execute the action of the selected rule
Remove the selected rule from the Executable Rules
The selected rule has remained **True** since its last execution

Figure 2.2: Control flow corresponding to the standard interpreter

In fact, any BR program in which the BRs have only monotonous actions, in the sense that the actions only affect variables which do not appear in the condition of any BR, can be executed by a simplified non-looping algorithm without changing the output. In particular, any direct translation of a decision tree (in which the leaves do not affect the tested attributes) into BRs can be executed on any input with a single check of each condition. This has led to the appearance of simplified execution modes in BRMS, such as the sequential execution mode in ODM, which uses the algorithm illustrated in Fig. 2.3:

- Order the elementary BRs according to their “priority” property

- Take each BR in order, and for each evaluate its condition then execute its action if the condition is True

Another example of such is the "FastPath" algorithm available as an execution mode in ODM, which simply evaluates the BRs' conditions all at once, then executes the actions of each BR r such that $T_r =$ True in an often preset arbitrary order. Such algorithms are popular in some industrial contexts as they simplify the BR engine to the point where the BR program does exactly what the non-programmer thinks it does, instead of doing so most of the time. However, BR languages defined using such algorithms are not as expressive as BR languages which use a main loop, as non-degenerate execution algorithms must evaluate an unbound number of evaluations of BR conditions, which leads to possible non-terminating inputs. In other words, non-looping execution algorithms are degenerate in the sense that they do not allow for any inference, whether forward-chaining or backwards-chaining.

An example of the execution a BR program using the basic interpreter $\mathscr{I}_0$ is described in Fig. 2.4, using the toy example of a BR program which takes as input the value amount of a requested loan (including interest), the yearly income of the requester and the duration (in years) of the repayment, then outputs either the original values (if the request is valid), an alternate loan which can be repaid, or an amount of zero if there is no possible repayment plan. This toy example supposes that the compound interest rate is 10% per year.

2.2 Machine Learning (ML)

The problem examined in this thesis, the parametrization of a BR program in light of a statistical objective, can be seen as a ML problem. The main purpose of using ML in our work is to provide an existing framework to examine our problem, as

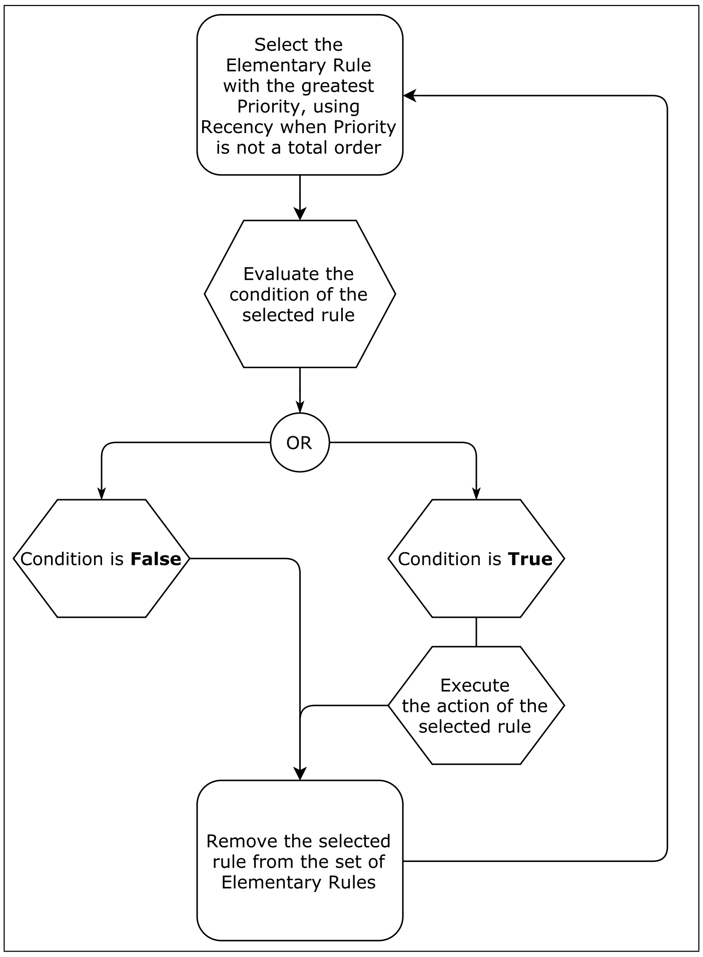

Select the Elementary Rule with the greatest Priority, using Recency when Priority is not a total order
Evaluate the condition of the selected rule
OR
Condition is **False**
Condition is **True**
Execute the action of the selected rule
Remove the selected rule from the set of Elementary Rules

Figure 2.3: Control flow corresponding to the sequential execution algorithm from ODM

well as use the known results pertaining to the learnability of function classes to prove that it cannot be solved in the general case.

Research linking Production Rules and Machine Learning has recently focused on transforming the product of ML techniques, such as decision trees or association rules, into executable production rules [124, 102, 11]. Past efforts have also used Production Systems as learning systems themselves, among other works on learning by doing [9, 141]. The ML problem of modifying an existing computer program or complex function to account for new data is sometimes called theory

The Rules (in order)

R_1 :

if (duration ≥ 10)
then (amount $\leftarrow 0$)

R_2 :

if (amount $\geq$ income $\times$ duration)
then (amount $\leftarrow 1.1 \times$ amount;
duration $\leftarrow$ duration $+ 1$)

The input Variables

amount	$\leftarrow 100$
income	$\leftarrow 10$
duration	$\leftarrow 9$

The Execution

Iteration 1:

Truth value of conditions:

$t(R_1) =$ False

$t(R_2) =$ True

Rules selected (in order): R_2

Rule executed: R_2

Variable values:

amount $= 110$

income $= 10$

duration $= 10$

Iteration 2:

Truth value of conditions:

$t(R_1) =$ True

$t(R_2) =$ True

Rules selected (in order): $R_1; R_2$

Rule executed: R_1

Variable values:

amount $= 0$

income $= 10$

duration $= 10$

Iteration 3:

Truth value of conditions:

$t(R_1) =$ False

$t(R_2) =$ False

Rules selected (in order) *None*

Rule executed: *None*

END

Figure 2.4: Example illustrating the execution algorithm: smallest loan payment plan

revision [51]. The theory revision problem restricted to modifying parameters of the existing theory is similar to our problem. An approach to the non-statistical theory revision applied to Business Rules appears in [31], working with CLIPS-R. Answers to the theory revision problem with access to labeled data points (with a known label for each input) appear in [4, 64, 65]. We work on extending work made on this problem to statistical goal learning problems in Production Rules systems, in particular BR systems.

In this section, we first formally express our problem and contrast it to existing ML formulations. We then present some existing results we will use in Chapter 3 pertaining to the learnability of function classes.

2.2.1 Statistical goal learning

One of the challenges of BR systems in the current industry is maintaining the flexibility that has lead to its adoption as the automatic decision management tool of choice by many organizations. While BRMS make it simple to maintain and modify BR programs, they do not offer a way to predict the effect of any modification, nor do they supply a framework for Machine Learning (ML) of BR programs in view of a goal. Business Users must rely on their knowledge of the business process and the goal to implement any modifications to the BRs on their own. In a digitalized world where ML and "big data" are becoming more than buzzwords, this can undermine the agility of BRMS, or at least make it costly and inefficient.

The goals of such modifications can be of a statistical nature, e.g. adjusting the average decision over a given set of inputs. For example, a bank might require that no more than a certain percentage, e.g. 30%, of all requested loans be examined manually, due to human resource concerns. As the number of manually examined loan requests should always be as high as possible, the goal is in fact to have the BR program which determines such things assign 30% of all loan requests to manual examination. Such a requirement could be fulfilled by automatically learning the "right values" of some adjustable parameters in the corresponding BR program. If the output of the BR program is a binary variable using 1 for manual evaluation and 0 for automatic evaluation of a loan request, the learning problem would be to find the values of the parameters which satisfy the statistical goal: "average of all outputs is 0.3". This could arise as a consequence of exceptional circumstances, e.g. a new legislation, or of natural trend evolution, e.g. the client base changing so that too many or too few loan requests are evaluated automatically.

Such a learning problem is what we henceforth call a statistical goal learning problem, as the goal is not over the output of the system but over the average of that output given a set of inputs. A broader definition of statistical goal learning goes beyond the average, and involves target statistical distributions in the outputs given a certain statistical distribution of inputs. An algorithm for the "narrower" version of the statistical goal learning problem will be discussed in Chapter 4. The broader version will be discussed in Chapter 5.

In the rest of this thesis, we characterize a BR program which must be learned this way by its input, output and parameters. The input is the initial value of the variables when executing the BR program, the output is the final value of the scalar variable chosen as the BR program's result and the parameters are the elements of the BRs which can be modified by the learning algorithm.

Definition 2.4 (Family of BR program)**.** We call *family of BR programs* derived from a BR program the set of functions $P(p, .)$ with values in $\mathbb{R}$, where $p \in \pi$ is a *parameter* or vector of parameters of the BRs, such that for any valid *input* $x \in X$ of the BR program, the *output* of the BR program modified by using the values p in place of predefined elements of BRs x is $P(p, x)$.

Let $P(p, .)$ be a family of BR programs parametrized over $p \in \pi \subseteq \mathbb{R}^{\phi}$, with input $x \in X \subseteq \mathbb{R}^d$. Let $g \in \mathbb{R}$ be the desired goal. The "narrower" statistical goal learning problem can then be formalized as:

$$\left.\begin{array}{rcl} & \min\limits_{p} \|p - p^0\| & \\ \left|\mathbb{E}_{q\in Q}\Big[f\Big(P(p, q)\Big)\Big] - g\right| & \leq & \varepsilon, \end{array}\right\} \tag{2.2}$$

where $\|\cdot\|$ is a given norm, $\mathbb{E}$ is the usual notation for the expected value, Q is a training set of inputs and ε is a given tolerance.

This formalization is closer to the reality of BR users than the alternative (minimizing $\mathbb{E}(P) - g$ while constraining $p - p^0$), as corporations will often consider goals more rigidly than changes to the business process, and the value of the objective will speak to them more as a kind of quantification of the changes to be made. The form this quantification takes, from minimizing the variation of each parameter in p to minimizing the number of parameters whose value is modified, depends on the definition of the norm $\|\cdot\|$. In terms of business modeling rules, this means that we can for example choose to try and change the rules minimally with many rules being changed, or we can try to change a minimal number of rules, even if the change is very big.

ML has many existing approaches and algorithms depending on what is to be learned and how it may be learned [80], the commonality being that ML aims to use training data in order to predict the behavior of other inputs, usually by learning a predictor function. In general, one may divide ML approaches into Supervised Learning and Unsupervised Learning [131], with problems of Semi-Supervised Learning straddling the line. In Supervised Learning, the training data is labeled, meaning the known data consists of both inputs and outputs of the function we wish to learn. In contrast, Unsupervised Learning only provides inputs, i.e. uses unlabeled training data and assumptions about the outputs (such as the number of classes in classification problems). In Semi-Supervised Learning problems, only some of the data is labeled. Those problems are usually studied as either imperfect Supervised Learning problems or constrained Unsupervised Learning problems.

Our problem lies somewhere between the two main approaches, as we do have information beyond the simple input data: we know both the original output of the BR program and the statistical distribution of the outputs we wish to achieve. The specificity of our problem however is that this information does not take the form of labels over data points, as in Supervised Learning. Instead, the information we have is closer to 'statistical labels', i.e. labels over data sets instead of data points. Just as in Unsupervised Learning, the relevance of the training data, usually historical data, lies solely in its statistical properties: the specific values are not important, as they are not individually labeled. The information we have about the desired distribution of outputs is also comparable to a (non-standard) assumption about the output made by Unsupervised Learning algorithms. Semi-Supervised Learning problems however are entirely unrelated to ours, since the problem is not one of partial labeling. While both the Supervised and Unsupervised approaches have been studied extensively, the Unsupervised Learning approach mostly focuses on classification problems, whether through clustering algorithms such as the k-means algorithm [47] or through association algorithms such as the Apriori algorithm [3]. Furthermore, using an Unsupervised approach would lead to disregarding the information encoded in the existing BR program. Such an approach is more applicable to initial learning of BRs, as some software (like IBM SPSS Modeler [117, 118]) can do in a limited fashion.

Consequently, similarities to our problem are better found among Supervised Learning problems. The traditional Supervised Learning problems are regression, for continuous outputs, and classification, for discrete outputs. The standard supervised learning algorithms are summarized by Hastie, Tibshirani and Friedman in [78]. The best known algorithms are Support Vector Machines [120], neural nets [84], logistic regression, naive bayes [106], k-nearest neighbors [67], random forests [104], and decision trees. Those methods, among others, are compared in [79] for the classification problem.

All of these methods consider the "classic" supervised learning problem, of which a generic formulation is the following, with a function f to be learned:

$$\min_{\hat{f}\in F} \left\| \left(\hat{f}(x) - y(x)\right)_{x\in X} \right\| \tag{2.3}$$

where F is a class of functions we are assuming contfains f (or a good approximation thereof), X is a finite learning set, $y(x)$ is the set of corresponding labels and $\|\cdot\|$ is a norm over the sequence of outputs, usually L_2 over $\mathcal{R}^X$. The different algorithms make different assumptions over F, whether it be the linearity of functions in it (linear regression), the countability of the output set (classification vs regression), or others. We assume in this thesis that all problems can be learned by using a class F taking the form of a set $\{f_p\}$, with p a vector of parameters.

The difference between the supervised learning problem and the proposed statistical goal learning problem lies in the constraint imposed by the known information: while the known information applies directly to the outputs in the "classic" problem, it is a statistical derivative of the outputs in the statistical problem. More precisely, the problem in Eq. 2.2 cannot be translated into the form of Eq. 2.3 because we do not have individual labels $y(x)$. The equivalent form would involve reworking the norm used in Eq. 2.3 to be a statistical measure of $\hat{f}$, which makes using standard supervised learning algorithms impractical for our purposes.

2.2.2 Computational Learning theory

Describing ML algorithms and their applicability to different problems is the purview of computational learning theory. There exist several approaches to this task, which can be broadly divided in two: those that specify that a successful learning algorithm must learn $\hat{f}$ exactly equal to f, and those that accept learning an approximation of f and thus use a definition of probability. In the first category, we find algorithmic learning theory [5] and exact learning [41], which both assume an unbounded (although finite) number of labeled samples to use. In the second category, we find well-known forms of learning theories such as the Vapnik—Chervonenkis theory [134], bayesian inference [14], and the Probably Approximatively Correct (PAC) framework [132], which is the one we use to prove our theorems.

A PAC learning algorithm identifies a function of the form $f : X \to \{0,1\}$ among a subset $\mathscr{C}$ of all such functions called a *concept class*. An algorithm $\mathcal{A}$ is an (ϵ, δ)-PAC learning algorithm for $\mathscr{C}$ if for any set of labeled data (S, Ω) of sufficient large size $\lambda = \texttt{card}(S)$, with $\omega_s \in \Omega$ the label corresponding to $s \in S$, the algorithms learns a function $h = \mathcal{A}(S, \Omega)$ such that:

$$\mathbb{P}\Big[\mathcal{A}(S,\Omega) = h \mid \mathbb{P}[h(x) \neq f(x) \mid x \in X] \leq \epsilon\Big] \geq 1 - \delta \tag{2.4}$$

- $\mathcal{A}$ is said to be efficient if the time complexity of $\mathcal{A}$ and h are polynomial in $1/\epsilon$; $1/\delta$; and λ.
- $\mathcal{A}$ is said to weakly learn $\mathscr{C}$ if there exist some polynomials $p_\epsilon(\lambda)$; $p_\delta(\lambda)$ for which $\epsilon \leq \frac{1}{2} - \frac{1}{p_\epsilon(\lambda)}$ and $\delta \leq 1 - \frac{1}{p_\delta(\lambda)}$.
- We say a concept class is PAC-learnable if it is both efficiently and weakly learnable. Otherwise, we say that the concept class is PAC-unlearnable.

This formalism translates to our statistical goal learning problem by using the correct concept class. In the rest of this thesis, we assume that the set $\mathscr{P}$ refers to a set of functions of the form $P(p, .)$ where P is a family of BR programs and $p \in \pi \subseteq \mathbb{R}^\phi$ parametrized this family. We consider the concept class whose members are indicator functions over $\mathscr{P}$, i.e. $c \in \mathscr{C}$ are functions such that $\forall p \in \pi, c(P(p, .)) \in \{0, 1\}$. We now wish to learn a specific member of $\mathscr{C}$: the mapping h which identifies BR programs satisfying the constraint from Eq. 2.2.

$$\forall p \in \pi, h(P(p, .)) = 1 \Leftrightarrow \Big|\mathbb{E}_{q \in Q}\Big[f\Big(P(p, q)\Big)\Big] - g\Big| \leq \varepsilon \tag{2.5}$$

Learning this concept leads naturally to learning the optimal parameter p solving Eq. 2.2, while being unable to learn it leads to the statistical goal learning problem being unsolvable. The problem of the learnability of the statistical learning problem is thus the problem of the existence of a PAC-algorithm $\mathcal{A}$ which can learn $\mathcal{C}$.

The broadest class of functions we know how to define formally is that of *Turing-computable* functions, as expressed by the Church-Turing thesis [66]. Pseudorandom functions (PRF), introduced by Goldreich, Goldwasser and Micali [50], are indexed families of functions F_p for which there exists a polynomial-time algorithm to evaluate $F_p(x)$, but no probabilistic polynomial-time algorithm can distinguish the function from a truly random function F_{rand} without knowing p, even if allowed access to an oracle. The existence of PRFs relies on the assumption of the existence of one-way functions, i.e. functions which can be evaluated in polynomial time but cannot be inverted in polynomial time [49]. This assumption is stronger than the usual $\mathsf{NP} \neq \mathsf{P}$ assumption.

It is known that PRF are not PAC-learnable [50, 34], which in turn means that the set of all Turing-computable functions is also not PAC-learnable. We use this result in Chapter 3.

2.3 Mathematical Programming (MP)

The approach we use to learn the parameters of BR programs is to treat the learning problem in Eq. 2.2 as an optimization problem. A common approach to optimization, which we adopt in this thesis, is called Mathematical Programming (MP) [142]. MP is the practice of using Mathematical Programs (or MP problems) to model and solve problems. Among those Mathematical Programs, we are make particular use of the Mixed-Integer Linear Programming (MILP) problems, which can be solved particularly well by existing optimization solvers.

In this section, we first introduce MP problems as well as some well-known classes of MP problems. We then mention the associated state of the art solvers and explain our choice of using CPLEX to test our algorithm (in Ch. 6).

2.3.1 MP formalism

An Optimization Problem can be written as:

$$\begin{aligned} &\underset{x}{\text{minimize}} && f(x) \\ &\text{subject to} && \\ &\quad \forall i \in \{1, \ldots, n_I\}, && g_i(x) \leq 0 \\ &\quad \forall i \in \{1, \ldots, n_E\}, && h_i(x) = 0 \\ & && x \in X \end{aligned}$$

where f, g_i and h_i are real-valued functions of the vector of decision variables x, $n_I \in \mathbb{N}$ and $n_E \in \mathbb{N}$ are respectively the number of inequality constraints and equality constraints in the problem, and X is a Cartesian product of intervals of $\mathbb{R}$ and finite subsets of $\mathbb{N}$. The function f is called the *objective* function. Any $x \in X$ is called a *solution*. If $x \in X$ satisfies $\forall i \in \{1, \ldots, n_I\}, g_i(x) \leq 0$ and $\forall i \in \{1, \ldots, n_E\}, h_i(x) = 0$, it is called a *feasible* solution. A feasible solution x^* is *optimal* if $f(x^*) \leq f(x)$ for all feasible solutions x.

A problem with no feasible solutions is called *infeasible*. Additionally, our problems are all derived from Eq. 2.2 in Sec. 2.2, which means the objective function f is always positive. In this thesis, we call *trivial* any optimization problem such that there exists an optimal solution x^* such that $f(x^*) = 0$. This corresponds to the optimization problems derived from trivial statistical learning problems, i.e. problems where the value of p^0 already satisfies the statistical goal.

Approaches for solving Optimization Problems are different for different classes of problems. The two most common criteria to classify MP problems are the existence of integrality constraints and the use of linear/convex/non-convex functions in the objective function and the constraints. Problems with integrality constraints such that all decision variables are integers are called Integer Programming problems. When some decision variables are continuous and some are discrete, the problem is a *Mixed-Integer Programming* (MIP) problem. Similarly, we call Linear Programming the branch of MP in which all the constraints g_i, h_i and the objective function f are linear, with Linear Programs often referring to non-integer linear programming problems. Quadratic Programming [89] and Convex Programming [32] are also well-researched branches of MP. The most general form of MP problems are nonconvex Mixed-Integer Nonlinear Programming (MINLP) problems.

In our thesis, we first describe a MIP formulation of the statistical goal programming problem from Sec. 2.2 for a subset of BR programs (Sec. 4.2), then we linearize the problem for a subset of even those (Sec. 4.4) in order to have a MILP problem which can be solved using standard solvers.

2.3.2 MP solvers

Solving MP problems can be done using exact methods, such as the simplex method for Linear Programming problems [115], or using converging approximations, such as the outer approximation method for Mixed-Integer Nonlinear Problems [39]. There are many algorithms for solving different MP problems. Algorithms based on heuristics perform the best in testing, although (as heuristics are bound to do) their exact performance depends on the MP problem which is being solved.

Different solvers implement different algorithms and recognize different classes of MP problems. We choose to use CPLEX as it is one of the state-of-the-art solvers for solving MILP problems [77] (the fact that it is developed by IBM at the France Lab, which is funding this thesis, is a bonus). Other commercial MILP solvers include FICO Xpress [42], Gurobi [52] and MOSEK [24]. Academic, non-commercial solvers are also competitive, including SCIP [111], CBC [25], GLPK [48] and LPSolve [73].

While our problems are too unwieldy to solve directly with general MIP solvers (a.k.a. MINLP solvers), especially as scalability is a concern for the applications of the statistical goal learning problem suggested in Sec. 2.1, MIP solvers are also available. The solvers SCIP, COUENNE [33], Antigone [7], Baron [12] and LindoGlobal [72] solve general nonconvex MINLP problems to global optimality. Others, such as Alpha-ECP [44] and Bonmin [17], guarantee global optimality only for convex problems, but can be used as heuristic solvers for nonconvex problems.

Chapter 3

BR programming language: Properties and learnability

In this chapter, we study the question of the learnability of BR programs, in the statistical goal learning sense. More precisely, we question the existence of an algorithm $\mathcal{A}$ which, using the notations from Subsec. 2.2.2, learns an indicator function h satisfying Eq. 2.5, in the PAC-learning sense of Eq. 2.4. We do so by first studying the BR programming language and its expressive power.

Any study of BRs as a programming language is meaningless without adding some constraints over the form of the conditions T and actions A of BRs, as we could perform any complex computations by phrasing them as a function otherwise. As an example, suppose we wish to program the arbitrary real-valued function f, a possible BR program when using the basic interpreter would be:

```
if switch = True then
    $x \leftarrow f(x)$
    switch $\leftarrow$ False
end if
```

In the rest of this thesis, unless otherwise mentioned we suppose that all conditions T and all actions A of BRs have a polynomial computational complexity. This assumption is realistic when considering industrially used BR programs, as automated decision making is by and large a task which is time sensitive and must be executed promptly.

In the first section, we prove that BR programs are Turing-complete when using an interpreter with a non-degenerate loop. We then provide an alternate, constructive proof and use it to describe a Structural Operational Semantics (SOS) over BR programs. We also showcase the usefulness of this semantics by demonstrating that the termination of some BR programs can be proven automatically using Prolog. Finally, we answer the main question of this chapter in the negative: our statistical goal learning problem is not PAC-learnable in the general case. In fact, it is even completely unlearnable, as we exhibit an example of a BR program which no algorithm can learn with a confidence rate of more than 50%, no matter how many oracle calls we allow.

3.1 Turing-Completeness of BR programs

Within the panorama of programming languages, one can distinguish two main categories: imperative languages and declarative languages. Since both categories contain Turing-complete languages, a separation of the two categories according to computational expressive power is impossible. On the other hand, by looking at three basic building blocks present in all imperative languages (assignments, tests and loops), we can informally segment programming languages more finely: purely imperative languages have explicit constructs for all three, and purely declarative languages do not have explicit constructs for any of those building blocks. However, programming languages can also fall somewhere between the two. One of the earliest computational models, lambda-recursive functions [69], is a declarative language which has explicit constructs for both assignments and tests. While it can be thought to embody loops right within the language itself by its recursion axiom, and tests and assignments have to somehow be "simulated". Prolog [99], Constraint Programming [10], and MP [142] are also declarative in that sense, in the sense that the language itself does not provide constructs of assignments, tests or loops. Perhaps the "purest" form of a purely declarative language is given by systems of Diophantine equations, famously shown to be Turing-complete when Hilbert's 10th problem was solved in the negative [75]. In this taxonomy, BRs taken as a programming language explicitly provides assignments and tests, but has no explicit loop construct.

The study of BR as a programming language starts with the fundamental question of its computational expressive power. The first thing we ask is whether BR programs are Turing-complete, i.e. whether we can decide if the execution of a BR program terminates at all. Asking the same question of any programming language amounts to asking whether the HALTING PROBLEM (HALT) can be solved on the class of Turing Machines (TM) that the programming language is able to describe, since HALT cannot be solved for Universal TMs (UTMs). Turing-completeness further provides an estimate of expressive power, since the widely accepted Church-Turing thesis [28, 128, 45] postulates that any effectively calculable function is Turing-computable. In other words, no device or program can compute a function that a UTM cannot.

Among the other programming languages we mentioned, recursive functions have been famously proved to be equivalent to Turing Machines (TM) by none other than Turing himself [126]; for Logic rules, including Prolog, we refer the reader to [116]. For MP, see [130].

3.1.1 Turing-Completeness

A Universal Turing Machine (UTM) is a TM which can simulate any other TM on arbitrary input [113, 127]. Let L be a programming language for the UTM U, described for example by its grammar. By means of a special program $\mathscr{I}$ called *interpreter*, programs written in L can be executed on U [76]. If a programming language L can be used to program all possible programs of a UTM via an interpreter, then L is said to be Turing-complete.

Definition 3.1 (Turing-completeness [127])**.** Let U be a UTM, which takes as input a string (T, x) consisting of a TM description and its input. A programming language L is *Turing-complete* if there exists an interpreter $\mathscr{I}$ such that for each possible input (T, x) of U there is a program p in L with $\mathscr{I}(p) = (T, x)$.

We can replace "UTM" in Defn. 3.1 by any universal computer described in any Turing-complete language L', since interpreters can be composed. More precisely: (i) let $\mathscr{I}'$ be the interpreter from L' to a UTM U', (T', x') be an input of U', and p' be a program in L' with $\mathscr{I}'(p') = (T', x')$; (ii) let $\mathscr{I}$ be an interpreter from L to L'. If there exists a program p in L such that $\mathscr{I}(p) = p'$, then $\mathscr{I}(\mathscr{I}'(p)) = (T', x')$ meaning L is a UTM.

Moreover, since a UTM is defined as a TM which is able to simulate any other TM, we prove a programming language L Turing-complete by showing that for any TM, L can be used to describe that TM via its interpreter, as was done in [35].

3.1.2 Basic Turing-completeness Proof

We now prove the Turing-completeness of BR programs in a two different ways: A straightforward construction of a UTM and a constructive reduction to WHILE programs. We wish to prove the expressive power of BR programs themselves, as opposed to the expressive power of the syntax allowed in the *conditions* and *actions* of BRs. For that reason, we maintain the restriction on the complexity of *conditions* and *actions* of BRs: they are at most polynomial. This avoids any simple but uninteresting BR programs which amount to:

```
if True then
    Solve the Halting problem
end if
```

Theorem 3.1 (Turing-completeness of BRs)**.** *The BR language restrained to polynomial complexity conditions and actions in BRs, when using the basic interpreter, is Turing-complete. Furthermore, any non-trivial BR interpreter involving a loop is also Turing-complete, as it can simulate the basic interpreter.*

We prove theorem 3.1 in a straightforward way by providing a BR program which simulates a UTM. We use the usual definition of a Turing Machine [129]. We note the states of a TM $q_1, \ldots, q_Q$, its tape symbols $s_1, \ldots, s_S$, its blank symbol s_b, its transition function $(q^i, s^i) \to (q^f, s^f, \mathsf{act})$ where $\mathsf{act} \in \{$"left", "right", "stay"$\}$, its initial state q^0, and its accepting states T_{er}. An initial tape T_0 is said to be accepted by a TM if the TM reading this tape stops and has a final state in T_{er}.

A BR program which simulates a UTM by being able to simulate any TM is described in Fig. 3.1. It uses the same notations, with initial values of $q = q^0$; of T a truncated T^0 containing a finite number of symbols, containing T^0_0 and containing all non-blank symbols of T^0; of $l = \text{size}(T)$; and of $p = 0$.

Some simplifications have been made for the sake of clarity: R_1 should clearly be at least three different rules each replacing act with one of "left", "right", "stay". The complete formally correct form would in fact have two more rules, in order to increase the length of the tape as needed, using the variable s_b.

Theorem 3.2. *the BR program described in Fig. 3.1 simulates any TM given an*

We suppose the variables include the following:

- many (static) state objects of type "state": $q_1, \dots, q_Q$
- many (static) symbol objects of type "symbol": $s_1, \dots, s_S$
- a (static) finite set of terminal states of type "terminal": T_{er}
- a (static) blank symbol of type "symbol": s_b
- a (static) set of Turing rules of type "rules", of the form $(\text{state}_{\text{initial}}, \text{symbol}_{\text{initial}}, \text{right}|\text{left}|\text{stay}, \text{state}_{\text{next}}, \text{symbol}_{\text{written}})$:
  $\mathcal{R} = \{(q_r^i, s_r^i, \mathsf{act}_r, q_r^f, s_r^f) \mid \mathsf{act}_r \in \{\text{"left"}, \text{"right"}, \text{"stay"}\}\}_r$
- the current state of type "state": q
- the length of the visible tape data, of type "length": l
- the current visible tape data of type "tape": $T = \{(i, s_i) \mid i \in \mathbb{N}, 0 \leq i \leq l-1\}$ where l is the length of the visible tape data
- the current place on the tape of type "position": p

We use the following meta-variables in the BR program that simulates a UTM:

- α_{q^f} of type "state"
- α_{s^f} of type "symbol"

The rule set to simulate a UTM is then written in a compact form:

```
R_1:
 if
   $(q, T(p), \mathsf{act}, \alpha_{q^f}, \alpha_{s^f}) \in \mathcal{R}$
 then
   $q \leftarrow \alpha_{q^f}$
   $T \leftarrow (T \setminus \{(p, T(p))\}) \cup \{(p, \alpha_{s^f})\}$
   $p \leftarrow p \pm 1$(Depending on the value of act)
   $l \leftarrow l \pm 1$(Depending on the respective values of act, $p$ and $l$)

R_2:
 if
   $(q \in T_{er})$
 then
   Stop;
```

Figure 3.1: A BR program which describes a UTM.

accepted tape. The final value of the tape and the final symbol of the TM will be identical to the final values of T and q.

Proof. We prove that the simulation is correct by induction over the number of steps n taken by the TM.

Before the TM takes any step ($n = 0$), its tape is identical to the value of T in the BR program before it executes any rule, as that tape is given to the BR as an input value. Similarly, the place on the tape at that point is the value of p and the state of the TM is equal to the value of q.

Assume that the tape, the position on it, and the state of the TM after step n are accurately represented by the BR program after n rule executions. We call T^{TM} the sequence representing the tape, p^{TM} the current place on the tape, and q^{TM} the current state of the TM.

If the TM halts, that means $\forall(\mathsf{act}, q^f, s^f), (q^{TM}, T^{TM}_{p^{TM}}, \mathsf{act}, q^f, s^f) \notin \mathcal{R}$. Thus, the BR does not execute R_1. Further, as the initial tape is accepted by the TM, we have $q^{TM} \in T_{er}$, which fulfills the condition for R_2. The BR program terminates at the same time as the TM, and its output is correct as R_2 does not modify values.

Otherwise, the TM will follow a rule in $\mathcal{R}$. Let us call it

$$r = (q^{TM}, T^{TM}_{p^{TM}}, \mathsf{act}, q^f, s^f).$$

In this case, the next BR executed will be R_1, and the only member of $\mathcal{R}$ to match will be r. In other words, the only relevant elementary rule is:

```
$R_1$:
 if        $(q, T(p), \mathsf{act}, q^f, s^f) \in \mathcal{R}$
 then      $q \leftarrow q^f$
           $T \leftarrow (T \setminus \{(p, T(p))\}) \cup \{(p, s^f)\}$
           $p \leftarrow p \pm 1$(Depending on the value of $\mathsf{act}$)
           $l \leftarrow l \pm 1$(Depending on the respective values of $\mathsf{act}$, $p$ and $l$)
```

because $q = q^{TM}$ and $T(p) = T^{TM}_{p^{TM}}$. As the action on this BR instance corresponds exactly to the modifications to the state and tape of the TM, the values stored on the tape of the TM after $n+1$ steps will again be the same as the values in the BR program. □

3.2 Constructive proof and resulting Operational Semantics

We propose an alternative proof of Turing-completeness, using a constructive simulator of WHILE programs in BR, since the Turing-Completeness of WHILE programs is well-known [55]. This proof underlines the relation between the BR language and standard imperative programming languages such as C or Java, which include a `while` instruction, which leads to establishing a canonical WHILE form of BR programs. We describe it in Subsec. 3.2.2.

As there exists a well-known Operational Semantics for WHILE programs [98], we can use this transformation from BR programs to WHILE programs to describe the Operational Semantics of BR programs. We characterize a Structural Operational Semantics for BR programs and use it to automatically decide the termination of some BR programs in Subsec. 3.2.3.

3.2.1 BR form of WHILE programs

A WHILE program has the canonical (recursive) form:

```
while $\mathsf{T}_0(x)$ do
    ifblock$_1(\mathsf{T}_1, \mathsf{A}_1, x)$
    $\dots$
    ifblock$_K(\mathsf{T}_K, \mathsf{A}_K, x)$
```

```
**end while**
```

where, for each $k \leq K$, **ifblock**$_k(\mathsf{T}_k, \mathsf{A}_k, x)$ is defined either as:

```
if $\mathsf{T}^1_k(x)$ then
    $x \leftarrow \mathsf{A}^1_k(x)$
    ifblock$(\mathsf{T}_k, \mathsf{A}_k, x)$
end if
```

or as an empty command. The interpretation of the symbols $\mathsf{T}^i_k(x)$ and $\mathsf{A}^i_k(x)$ is: T are tensors of Boolean conditions on the variables x, which evaluate to True or False, and A is a tensor of functions of x yielding values to be assigned to the variables.

In other words, a WHILE program is a single conditional loop containing a sequence of embedded test conditions followed by a conditional assignment action.

We prove the Turing-completeness of BR programs by showing that a generic WHILE program can be interpreted into a BR program. The only requirement of the interpretation is to be computable. We first prove this for WHILE programs without embedded **if** statements, then we exhibit a sequence of syntactical steps on the symbols of a generic WHILE program which transforms it into a WHILE program without embedded **if** statements. As WHILE programs are themselves Turing-complete [55], this proof also shows that BR programs and WHILE programs compute the same functions.

Lemma 3.3. *Any WHILE program without embedded* **if** *statement can be simulated using a BR program.*

Proof. Given the following WHILE program without embedded **if** statements:

```
1: while $\mathsf{T}_0(x)$ do
2:     if $\mathsf{T}_1(x)$ then
3:         $x \leftarrow \mathsf{A}_1(x)$
```

```
4:     end if
5:     ...
6:     if $\mathsf{T}_K(x)$ then
7:         $x \leftarrow \mathsf{A}_K(x)$
8:     end if
9: end while
```

We can write an equivalent rule set with $2\,K-1$ rules and an additional variable c, which takes as value the index of the last action A_k executed and has initial value 0:

```
if $c < 2 \wedge \mathsf{T}_2(x_1, ..., x_n)$ then
    $x \leftarrow \mathsf{A}_2(x)$
    $c \leftarrow 2$
end if
...
if $c < K \wedge \mathsf{T}_K(x_1, ..., x_n)$ then
    $x \leftarrow \mathsf{A}_K(x)$
    $c \leftarrow K$
end if
if $\mathsf{T}_0(x_1, ..., x_n) \wedge \mathsf{T}_1(x_1, ..., x_n)$ then
    $x \leftarrow \mathsf{A}_1(x)$
    $c \leftarrow 1$
end if
if $\mathsf{T}_0(x_1, ..., x_n) \wedge \mathsf{T}_2(x_1, ..., x_n)$ then
    $x \leftarrow \mathsf{A}_2(x)$
    $c \leftarrow 2$
end if
...
```

```
**if** $\mathsf{T}_0(x_1, ..., x_n) \wedge \mathsf{T}_K(x_1, ..., x_n)$ **then**
    $x \leftarrow \mathsf{A}_K(x)$
    $c \leftarrow K$
**end if**
```

We prove by induction that this BR program using the basic interpreter executes the same assignment actions as the WHILE program.

As the initial value of c is $c = 0$, the first action executed by the BR program is the assignment $x \leftarrow \mathsf{A}_k(x)$ such that k is the smallest integer satisfying $\mathsf{T}_0(x_1, ..., x_n) \wedge \mathsf{T}_k(x_1, ..., x_n) = \mathsf{True}$. If $\mathsf{T}_0(x_1, ..., x_n) = \mathsf{False}$, then there is no such k and the BR program does nothing, which is the same as the WHILE program since it cannot enter the loop at all. Suppose $\mathsf{T}_0(x_1, ..., x_n) = \mathsf{True}$. If k does not exist, both the WHILE program and the BR program do nothing. If k exists, then we have:

$$\mathsf{T}_k(x_1, ..., x_n) = \mathsf{True}$$
$$\text{and} \quad \forall h < k, \mathsf{T}_h(x_1, ..., x_n) = \mathsf{False}$$

which is identifies exactly the first if-then statement to be executed in the WHILE program.

We now assume at least one assignment was executed in the WHILE program, and that the actions executed by the BR program are exactly those executed by the WHILE program so far. The value of c is the index of the last executed action. Let $k \leq K$ be such that the next action executed by the WHILE program is $x \leftarrow \mathsf{A}_k(x)$.

If k does not exist, we have:

$$\mathsf{T}_0(x_1, ..., x_n) = \mathsf{False}$$
$$\text{and} \quad \forall h > c, \mathsf{T}_h(x_1, ..., x_n) = \mathsf{False}$$

which means that each *condition* in the BR program is False. Both the WHILE program and the BR program halt at this point.

If $k > c$, we have:

$$\begin{aligned} \mathsf{T}_k(x_1, ..., x_n) &= \mathsf{True} \\ \text{and} \quad \forall c < h < k, \mathsf{T}_h(x_1, ..., x_n) &= \mathsf{False} \end{aligned}$$

because the WHILE program reaches the assignment indexed by k without entering any if-then block in between. This with the definition of c means the first rule in the BR program to have a True *condition* is:

```
if $c < k \wedge \mathsf{T}_k(x_1, ..., x_n)$ then
    $x \leftarrow \mathsf{A}_k(x)$
    $c \leftarrow k$
end if
```

and the next assignment to be executed in the BR program is the same as the one in the WHILE program: $x \leftarrow \mathsf{A}_k(x)$.

If $k \leq c$, we have:

$$\begin{aligned} \mathsf{T}_0(x_1, ..., x_n) &= \mathsf{True} \quad (1) \\ \text{and} \quad \mathsf{T}_k(x_1, ..., x_n) &= \mathsf{True} \quad (2) \\ \text{and} \quad \forall c < h, \mathsf{T}_h(x_1, ..., x_n) &= \mathsf{False} \quad (3) \\ \text{and} \quad \forall h < k, \mathsf{T}_h(x_1, ..., x_n) &= \mathsf{False} \quad (4) \end{aligned}$$

because the WHILE program reaches the end of the loop, passes the looping test, then reaches the assignment indexed by k without entering any if-then block in between. The conditions of the first $K - 1$ BRs in the BR program are evaluated to False because of (3) and the value of c. The equalities (1), (2) and (4) combine to show that the first rule in the BR program to have a True *condition* is:

```
if $\mathsf{T}_0(x_1, ..., x_n) = \mathsf{True} \wedge \mathsf{T}_k(x_1, ..., x_n)$ then
```

```
    $x \leftarrow \mathsf{A}_k(x)$
    $c \leftarrow k$
end if
```

and the next assignment to be executed in the BR program is again the same as the one in the WHILE program: $x \leftarrow \mathsf{A}_k(x)$.

The BR program and the WHILE program execute exactly the same assignment instructions, they are thus equivalent. $\square$

Lemma 3.4. *Any WHILE program can be transformed into an equivalent WHILE program without embedded* **if** *statements.*

Proof. We prove the lemma by reasoning on the **ifblocks**. We consider the assertion: Any **ifblock** can be transformed into a finite sequence of **ifblocks** without embedded **if** statements.

We reason inductively on the depth of the deepest embedded **if** statement. If it is 0 or 1, the property is trivial (one is a pass instruction and the other already of the correct form).

Let $n \in \mathbb{N}$ such that we can transform any **ifblock** of depth n into a sequence of **if** statements. Let us consider an **ifblock** of depth $n + 1$. It can be written as:

```
if $\mathsf{T}_1(x)$ then
    $x \leftarrow \mathsf{A}_1(x)$
    if $\mathsf{T}_2(x)$ then
        $x \leftarrow \mathsf{A}_2(x)$
        ifblock($\mathsf{T}_3, \mathsf{A}_3, x$)
    end if
end if
```

where **ifblock**(T_3, A_3, x) is an **ifblock** of depth $n - 1$. It is equivalent to the following:

```
if $\mathsf{T}_1(x) \wedge \mathsf{T}_2(x)$ then
    $x \leftarrow \mathsf{A}_2(\mathsf{A}_1(x))$
    ifblock$(\mathsf{T}_3, \mathsf{A}_3, x)$
end if
if $\mathsf{T}_1(x) \wedge \neg\mathsf{T}_2(x)$ then
    $x \leftarrow \mathsf{A}_1(x)$
end if
```

Which is a sequence of two **ifblocks** of depth n. As we can tranform each of them into a sequence of **ifblocks** without embedded **if** statements, we have the property for **ifblocks** of depth $n+1$, and the induction is valid.

The property applied to each **ifblocks** of a WHILE program transforms it into the form we wanted. □

We can now prove Th. 3.1 constructively, as any WHILE program can be transformed into a WHILE program without embedded ifs and thus into a BR program. This proves that BR programs can compute any WHILE program. Since WHILE programs are Turing-complete [55], BR programs are indeed Turing-complete.

3.2.2 WHILE form of BR programs

We can now extend this constructive proof to the existence of a canonical WHILE form of BR programs for the basic interpreter $\mathscr{I}_0$. The Fig. 3.2 shows the WHILE form of the example in Fig. 2.4.

Theorem 3.5 (WHILE form of BRs)**.** *Any BR program executed using the basic interpreter $\mathscr{I}_0$ can be written as a WHILE program using a canonical transformation.*

Proof. A canonical WHILE program equivalent to a BR program is easy to establish when using the basic execution algorithm. The main idea is to introduce an additional integer variable x_0 to serve as a control variable, and to write each elementary rule explicitly in the WHILE program itself.

For a BR program in which there are ρ BRs called $R_1, \ldots, R_\rho$ with *conditions* $T_1, \ldots, T_\rho$ and *actions* $A_1, \ldots, A_\rho$ and the variable is x, the WHILE program equivalent to the BR program executed with the basic interpreter $\mathscr{I}_0$ is written as below. It uses a single additional integer variable x_0.

```
1:  $x_0 \leftarrow -1$
2:  while $x_0 \neq 0$ do
3:      if $x_0 = -1$ then
4:          if $T_1(x) = \mathsf{True}$ then
5:              $x_0 \leftarrow 1$
6:          else if ... then
7:              ...
8:          else if $T_\rho(x) = \mathsf{True}$ then
9:              $x_0 \leftarrow \rho$
10:         else
11:             $x_0 \leftarrow 0$
12:         end if
13:     else if $x_0 = 1$ then
14:         $x \leftarrow A_1(x)$
15:         $x_0 \leftarrow -1$
16:     else if ... then
17:         ...
18:     else if $x_0 = \rho$ then
19:         $x \leftarrow A_\rho(x)$
```

```
        $x_0 \leftarrow -1$
    else
        $x_0 \leftarrow 0$
    end if
end while
```

This WHILE program is well-defined and obviously simulates the execution of the BR program, with two consecutive iteration of the while loop corresponding to one iteration of the BR program's execution loop. □

The input Variables

amount	$\leftarrow 100$
income	$\leftarrow 10$
duration	$\leftarrow 9$

The WHILE program

```
$x_0 \leftarrow -1$
while $x_0 \neq 0$ do
    if $x_0 = -1$ then
        if duration $\geq 10$ then
            $x_0 \leftarrow 1$
        else if amount $\geq$ income $\times$ duration then
            $x_0 \leftarrow 2$
        end if
    else if $x_0 = 1$ then
        amount $\leftarrow 0$
        $x_0 \leftarrow -1$
    else if $x_0 = 2$ then
        amount $\leftarrow 1.1 \times$ amount
        duration $\leftarrow$ duration $+ 1$
        $x_0 \leftarrow -1$
    else
        $x_0 \leftarrow 0$
    end if
end while
```

Figure 3.2: WHILE form of the rule set in Fig. 2.4

3.2.3 A Structural Operational Semantics for BR programs

Since any BR program can be rewritten as a WHILE program, the WHILE form of BR programs can be used to describe a clear and well-known Structural Operation Semantics (SOS) over BR programs. Let us consider the SOS described by Plotkin in [98] applied to WHILE programs. It describes the execution of a program as a finite automaton, with each simple command corresponding to a state transition relation. Its usefulness in proving properties of programs, such as sets of input resulting in termination or non-termination, has been established [22]. Fig. 3.3 shows the state transition relations corresponding to the WHILE language under an easily readable form, using a Logic Rule syntax wherein the premises are listed above a horizontal line, the conclusion below, and the condition, if present, to its right.

Termination of a program for a given input is one of the easiest properties of programs to look into using SOS. Because of the Turing-completeness of the BR language, our methodology for proving (or disproving) termination will obviously not give results on every BR program.

We use an example to demonstrate that the transformation of a BR program into a WHILE program can be used to prove properties of BR programs through those same SOS methods. Consider the simple example of a loan request application with three integer variables: `amount`, `duration` and `income`. We use the naive rule described in Fig. 3.4. The input value of `amount` is the requested loan value, while its final value is the total repaid sum.

We use Prolog [99] to code the SOS of the WHILE programming language as rules in a rule inference engine using the form displayed in Fig. 3.3, with similar rules encoding the evaluation of Boolean and Numerical variables. The choice of Prolog is due to the way it allows for both natural encoding of logical inference rules and clearly formulated complex queries. The latter are used in our case to describe the WHILE program and the set of inputs for which the termination of the BR program is being investigated. The existence of non-terminating Prolog programs [100] is no obstacle to this application, since it is known that proving the termination of a given program is an undecidable problem [126].

In our example, we encode the WHILE program itself as a Prolog term[1]. Asking Prolog about the termination of this program is as simple as querying about the

[1]Notably, we use the variables $X = Amount \times 10^i$ and $Y = income \times 10^i$ where i is the number of times the rule executes, so as to benefit from Prolog's CLP(FD) library [125].

Supposing C are commands, E are expressions and s are memory states, the following logic rules encode the Operational Semantics of the commands of the WHILE language.

The assignment command:

$$\frac{< E, s > \to_e < E', s' >}{< \mathtt{x} \leftarrow E, s > \to_c < \mathtt{x} \leftarrow E', s' >}$$

$$\frac{}{< \mathtt{x} \leftarrow \mathtt{n}, s > \to_c < \mathtt{skip}, s[\mathtt{x} = \mathtt{n}] >}$$

The sequential composition command:

$$\frac{< C_1, s > \to_c < C_1', s' >}{< (C_1; C_2), s > \to_c < (C_1'; C_2), s' >}$$

$$\frac{}{< (\mathtt{skip}; C_2), s > \to_c < C_2, s >}$$

The `if` command:

$$\frac{}{< \mathtt{if}\ \mathsf{True}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < C_1, s >}$$

$$\frac{}{< \mathtt{if}\ \mathsf{False}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < C_2, s >}$$

$$\frac{< B, s > \to_b < B', s' >}{< \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < \mathtt{if}\ B'\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s' >}$$

The `while` command:

$$\frac{}{< \mathtt{while}\ B\ \mathtt{do}\ C, s > \to_c \left\langle \begin{array}{ll} \mathtt{if}\ B\ \mathtt{then} & (C; \mathtt{while}\ B\ \mathtt{do}\ C) \\ \quad \mathtt{else} & \mathsf{skip} \end{array}, s \right\rangle}$$

Figure 3.3: The logic rules encoding the Operational Semantics of commands in WHILE, starting from a state s of the machine's memory

truth value of the set of assertions $\{X0\_fin = 0, X\_in = Amount, Y\_in = 10, Dur\_in = 5\}$.

The general procedure to check the termination of a BR program is clearly shown by the example: given a BR program, we transform it into its WHILE

The Rule

R:

if (amount $\geq$ income $\times$ duration)
then (amount $\leftarrow 1.1 \times$ amount,

duration $\leftarrow$ duration $+ 1$)

The WHILE program

```
$x_0 \leftarrow -1$
while $x_0 \neq 0$ do
    if $x_0 = -1$ then
        if amount $\geq$ income $\times$ duration then
            $x_0 \leftarrow 1$
        else
            $x_0 \leftarrow 0$
        end if
    else if $x_0 = 1$ then
        amount $\leftarrow 1.1 \times$ amount
        duration $\leftarrow$ duration $+ 1$
        $x_0 \leftarrow -1$
    else
        $x_0 \leftarrow 0$
    end if
end while
```

The Prolog encoding

We use the variables $X = \textit{amount} \times 10^i$ and $Y = \textit{income} \times 10^i$ where i is the number of times the rule executes.

```
C = seq(C1,C2)
C1 = assign('X0',-1)
C2 = whiledo(neq('X0',0),C3)
C3 = ifthenelse(eq('X0',-1),C4,C5)
C4 = ifthenelse(geq('X',mult('Y','Dur')),
        assign('X0',1),assign('X0',0))
C5 = ifthenelse(eq('X0',1),seq(seq(seq(
        assign('Dur',add('Dur',1)),
        assign('X',mult('X',11))),
        assign('Y',mult('Y',10))),
        assign('X0',-1)),
        assign('X0',0))
```

Figure 3.4: A naive BR program for Loan Applications

form, which we then use to derive the SOS. We implement the SOS as a set of constraints in a Prolog dialect, and finally we use the corresponding interpreter to try and establish feasibility or infeasibility of the constraint set.

We coded the WHILE programs for three examples in Prolog. We then used *SWI-Prolog* [121] to test termination for an interval range of inputs by using a Prolog query containing the input and the specific output $x_0 = 0$ (i.e. the WHILE loop ends). The results from Fig. 3.7 for the BR program in Fig. 3.4 show some of the advantages and limits of this method. For some intervals of values for *amount*, we can prove in a few seconds of CPU time that for *duration* $= 5$ and *income* $= 10$, the BR program always terminates. On the other hand, for this particular BR program and variable, we are unable to prove non-termination on an interval. We remark, however, that the most useful contribution of an SOS is to prove termination of programs (rather than non-termination). Amongst its many advantages, our semantics can be used for automatic validation of a BR program, for example. In our example, a search algorithm could also be used to identify the cutoff point of *amount* ≤ 62 as containing every terminating input, if one were to know of the existence of such a cutoff point.

Along with the example from Fig. 3.4, a slightly more realistic example is given in Fig. 3.5, adding the boolean variable approval and the integer variable age to the BR program. On the other hand, Fig. 3.6 is a less trivial example: a BR program that chooses the interest rate interest depending on the credit score score using a made-up algorithm which simulates the logistic map, where score $\in [300, 850]$.

The Rules

R_1:
  if (approval = True $\wedge$ age < 18)
  then (approval $\leftarrow$ False)

R_2:
  if (approval = True $\wedge$ duration > 10)
  then (approval $\leftarrow$ False)

R_3:
  if (approval = True $\wedge$ amount $\geq$ income $\times$ duration)
  then (amount $\leftarrow 1.1 \times$ amount,
  duration $\leftarrow$ duration $+ 1$)

The input Variables

boolean approval = True
int age, duration, income
float amount

The WHILE program

```
$x_0 \leftarrow -1$
while $x_0 \neq 0$ do
    if $x_0 = -1$ then
        if approval = True $\wedge$ age $< 18$ then
            $x_0 \leftarrow 1$
        else if approval = True $\wedge$ duration $> 10$ then
            $x_0 \leftarrow 2$
        else if approval = True $\wedge$ amount $\geq$ income $\times$ duration then
            $x_0 \leftarrow 3$
        else
            $x_0 \leftarrow 0$
        end if
    else if $x_0 = 1$ then
        approval $\leftarrow$ False
        $x_0 \leftarrow -1$
    else if $x_0 = 2$ then
        approval $\leftarrow$ False
        $x_0 \leftarrow -1$
    else if $x_0 = 3$ then
        amount $\leftarrow 1.1 \times$ amount
        duration $\leftarrow$ duration $+ 1$
        $x_0 \leftarrow -1$
    else
        $x_0 \leftarrow 0$
    end if
end while
```

Figure 3.5: A more realistic BR program for Loan Applications, the input should always have approval = True

The results for the examples in Fig. 3.4 and Fig. 3.5 are displayed in Fig. 3.7, while the results for the example in Fig. 3.6 are in Fig. 3.8. Fig. 3.7 is indicative of a practical use of studying the termination of BR programs this way: a bank might wish to check that their loan application BR program terminates for a given range of values. Studying unexpected failures can point out a missing rule or variable in the BR program.

The Rules

R_1:
if $(n \leq 1000)$
then $(x \leftarrow r \cdot x(1-x))$

R_2:
if $(x \geq 0.51)$
then $(x \leftarrow r \cdot x(1-x))$

R_3:
if $(x \leq 0.49)$
then $(x \leftarrow r \cdot x(1-x))$

The Variables

float $r = 4\frac{\mathsf{score}}{850} \in [0,4]$
float $x = 0.48 + \mathsf{interest} \in [0,1]$
int $n = 1$

The WHILE program

```
$x_0 \leftarrow -1$
**while** $x_0 \neq 0$ **do**
    **if** $x_0 = -1$ **then**
        **if** $n \leq 1000$ **then**
            $x_0 \leftarrow 1$
        **else if** $x \geq 0.51$ **then**
            $x_0 \leftarrow 2$
        **else if** $x \leq 0.49$ **then**
            $x_0 \leftarrow 3$
        **else**
            $x_0 \leftarrow 0$
        **end if**
    **else if** $x_0 = 1$ **then**
        $x \leftarrow a \cdot x(1-x)$
        $x_0 \leftarrow -1$
    **else if** $x_0 = 2$ **then**
        $x \leftarrow a \cdot x(1-x)$
        $x_0 \leftarrow -1$
    **else if** $x_0 = 3$ **then**
        $x \leftarrow a \cdot x(1-x)$
        $x_0 \leftarrow -1$
    **else**
        $x_0 \leftarrow 0$
    **end if**
**end while**
```

Figure 3.6: A nontrivial ruleset and corresponding WHILE program, the input should always have $n = 1$

On the other hand, the results from Fig. 3.8 are somewhat more difficult to interpret. While the BR program in Fig. 3.6 is somewhat contrived, it shows that some failures can be not only unexpected, but unpredictable. The `ERROR: Out of local stack` looks like yet another inconclusive answer, but tracing the execution of the Prolog interpreter shows that these errors are the result of infinitely repeated goals, and a simple alteration to the Prolog interpreter can detect most infinite recursion loops of that kind [133]. In other words, a slight modification of the Prolog interpreter leads to our method providing a successful non-termination proof in cases like this one. These are non-terminating inputs that might be found by looking at the internal states of a classic BR engine, but the automation provided by existing research on SOS and Prolog remains valuable and time-saving. The BR program in Fig. 3.6 actually simulates the fixed points of the logistic map, defined as the sequence $x_{n+1} = r \cdots x_n(1 - x_n)$, and falls into a infinite recursion of goals if no fixed point is within $[0.49, 0.51]$. The bifurcation diagram in Fig. 3.9

is well-known, and represents those fixed points. While it makes the situation clear for $r \in [1.41, 3.57]$, i.e. $\mathsf{score} \in [300, 758]$, the following values fall within the chaotic part of the logistic map. In particular, some specific values have a periodic behavior, such as the tested $r = 3.82843$ ($\mathsf{score} = 813.541375$). The use of SOS derived from the WHILE form of BR programs helps in this and similar cases by automatically identifying with certainty non-terminating inputs, assuming Prolog is configured to identify infinite recursive goals.

Methods exist to analyze the termination of Prolog programs [101, 135]. However, I do not believe the termination of the Prolog program itself is of interest to our case, as there are many infinite data structures that can be created by queries constructed by our method and they do not all have the same meaning for the termination of the BR program itself. Notably, either of the errors in Figures 3.7 and 3.8 stem from infinite data structures, but only the error in Figure 3.8 indicate non-termination of the input in question – although there is in fact a non-terminating input in the range $60 \leq \mathsf{amount} \leq 70$ from Figure 3.7, that cannot be discerned from the execution of the Prolog program.

Input range	$\mathsf{amount} \leq 50$	$50 \leq \mathsf{amount} \leq 60$	$60 \leq \mathsf{amount} \leq 70$	$\mathsf{amount} \leq 62$	$63 \leq \mathsf{amount}$
Naive program (Fig. 3.4)	True	True	`ERROR:` `'Out of global stack'`	True	`ERROR:` `'Out of global stack'`
Realistic program	True	True	True	True	True

Figure 3.7: Results of the Prolog query containing the SOS of WHILE forms of the BR programs in Fig. 3.4 and Fig. 3.5 as well as facts about the input (duration = 5, income = 10, and age = 20 when relevant) and about the output ($x_0 = 0$)

Input value	r = 3	r = 3.22	r = 3.67	r = 3.82843
Equivalent credit score	score = 637.5	score = 684.25	score = 779.875	score = 813.541375
Query result	`ERROR: Out of local stack`	True	True	`ERROR: Out of local stack`

Figure 3.8: Results of the Prolog query containing the SOS of WHILE forms of the BR programs in Fig. 3.6 as well as facts about the input ($x = 5$, $n = 1$) and about the output ($x_0 = 0$) All values in this table are exact values

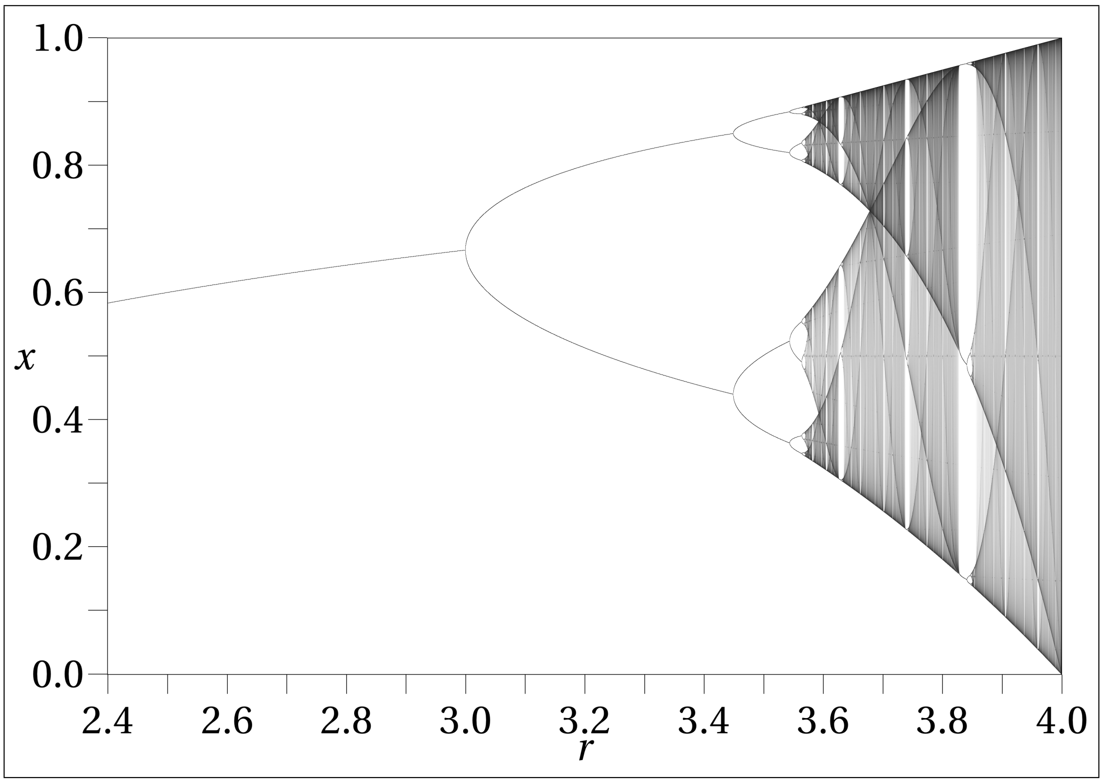


Figure 3.9: Bifurcation Diagram for the Logistic Map

3.3 Learnability of the statistical goal learning problem

In this section, we first use the Turing-completeness of BR programs to prove that the learning problem in Eq. 2.2 is not PAC-learnable. Furthermore, we then use the same example of a known chaotic map, the logistic map [26], to prove that the statistical learning problem is completely unlearnable in the general case.

3.3.1 PAC-Learnability

The exploration of BRs as a programming language leads to some basic insight related to the learnability of BR programs in the general case. The Turing-completeness of BRs means that HALT is undecidable for BR programs, thus making any learning problem over the general class of all BR programs unsolvable. In the rest of this thesis, we thus limit ourselves to learning problems over classes of BR programs known to terminate.

Pseudorandom functions (PRF), introduced by Goldreich, Goldwasser and Micali [50], are indexed families of functions F_p for which there exists a polynomial-time algorithm to evaluate $F_p(x)$, but no probabilistic polynomial-time algorithm can distinguish the function from a truly random function F_{rand} without knowing p, even if allowed access to an oracle. It is known that PRF are unlearnable using PAC algorithms [50, 34].

As PRFs can be described by Turing-complete programming languages, the fact that BR programs in general are not PAC-learnable is trivial.

We now consider the PAC-learnability problem from Eq. 2.4 in Sec. 2.2 in the context of our statistical goal learning problem. We have $(P(p,.))_{p\in\pi}$ a class of terminating BR programs indexed by p, Q a list of items from the input domain X with $\text{card}(Q) = \lambda$, g a goal for the value of the average output $\mathbb{E}_{q\in Q}[f(P(p,q))]$, and ϵ the tolerance for the learning problem. We consider $\mathscr{C}$ the concept class whose members are $c : (\mathscr{P}_p)_{p \in \pi} \to \{0, 1\}$.

We now wish to learn a specific member of $\mathscr{C}$: the mapping h defined in Eq. 2.5, which outputs 1 iff the BR program $P(p, .)$ satisfies the constraint from Eq. 2.2: $\left|\mathbb{E}_{q \in Q}\left[f\left(P(p, q)\right)\right] - g\right| \leq \varepsilon$. Learning this concept leads naturally to learning the optimal parameter p solving Eq. 2.2, while being unable to learn it leads to Eq. 2.2 being unsolvable.

Theorem 3.6. *The concept class $\mathscr{C}$ is unlearnable: specifically, the concept $h \in \mathscr{C}$ defined as $h(p) = 1$ iff $\left|\mathbb{E}_{q \in Q}\left[f\left(P(p, q)\right)\right] - g\right| \leq \varepsilon$ cannot be learned using a PAC learning algorithm in the general case.*

In other words, there is no practically viable algorithm that can learn a BR program out of a class of BR programs in the general statistical goal learning case, even with access to a perfect oracle. This is a consequence of both the Turing-completeness of BR programs and the unlearnability of PRF.

Proof. As BR programs are Turing-complete, we choose the family $P(p, .)$ to be a PRF. Any algorithm that learns $\mathscr{C}$ also learns $(\mathbf{1}_f(p))_p \subset C$, where $\mathbf{1}_f(p) = 1$ iff $\mathscr{P}_p = f$. Learning the latter is trivially the same as learning a PRF, which is proven to be impossible. $\square$

The specific example mentioned in the theorem answers the question of PAC-learnability for the statistical BR learning problem in the general case. Even if the concept we wish to learn is described more broadly than by providing an oracle for a specific BR program, it is impossible to adjust the statistical behavior of BR programs according to a predefined goal.

3.3.2 Complete unlearnability

While PAC-learnability has been disproved, we also prove that there exist Turing-computable functions which suffer from actual unlearnability – in the sense that no PAC algorithm can learn a concept class, whether or not that algorithm learns it weakly or efficiently. This is stronger than what was proved for PRFs in [34]. We have used the fact that PRFs are not PAC-learnable in the sense that no PAC algorithm can efficiently and weakly learn a PRF. We now demonstrate an example of a concept class that cannot be learned by PAC algorithms at all, thus proving that Turing-complete languages cannot be learned by PAC algorithms in the general sense. This example is based on the intuition that chaos cannot be predicted, and so cannot be learned. This in turn also means that the statistical learning problem cannot be learned by PAC algorithms in the general case.

We use a known chaotic map, the logistic map $f_{n+1}(x) = af_n(x)(1 - f_n(x))$, $f_0(x) = x$, with the parameter $a = 4$. Some of its properties are presented by Berliner [26]. We call $(C_n)_{n\in\mathbb{N}}$ the concept class such that $\forall x \in X = [0, 1], C_n(x) = 1$ iff $f_n(x) \geq 0.5$ and $C_n(x) = 0$ otherwise, where x follows the arcsine distribution, i.e. the probability density function is $p(x) = \frac{1}{\pi\sqrt{x(1-x)}}$, and n follows the uniform distribution.

Theorem 3.7. *The concept class $(C_n)_n$ cannot be learned with any accuracy. To be precise, for all algorithms $\mathcal{A}$ calling the oracle $C_n(x)$ a finite number of times, we have:*

$$\mathbb{P}_{n\in\mathbb{N}}(\mathbb{P}_{x\in X}(\mathcal{A}(C_n)(x) \neq C_n(x)) = 0.5) = 1$$

Proof. The proof relies heavily on Berliner's paper [26]. From it, we know that as the logistic map is chaotic, each sequence $(f_n(x))_n$ is either eventually periodic or is dense in $[0, 1]$. We also know that as X follows the arcsine distribution, the $C_n(X)$ are i.i.d. Bernoulli random variables, such that $\mathbb{P}_{x\in X}(C_n(x) = 1) = 0.5$.

Suppose $\mathcal{A}$ calls $C_n(x)$ for values of $x \in \{x_1, \dots, x_k\}$. We call n^0 the value such that $\mathcal{A}(C_n) = C_{n^0}$. As $\mathbb{P}_{x \in X}(C_{n^0}(x) = 1) = 0.5$ does not depend on n^0, and the $C_n(X)$ are i.i.d., we have $\mathbb{P}_{x \in X}(C_{n^0}(x) \neq C_n(x)) = 0.5$ iff $n^0 \neq n$ and $\mathbb{P}_{x \in X}(C_{n^0}(x) \neq C_n(x)) = 0$ otherwise. The theorem is thus the same as saying that $\mathcal{A}$ almost certainly (in the probabilistic sense) cannot match n^0 to the exact value of n. We now prove that there almost always exists $n^1 \neq n$ which is indistinguishable from n by $\mathcal{A}$, i.e. $C_{n^1}(x_1) = C_n(x_1), \dots, C_{n^1}(x_k) = C_n(x_k)$.

Let us call $Y_i^1 = C_i(x_1), \dots, Y_i^k = C_i(x_l)$ with $i \in \mathbb{N}$. Some of the sequences Y^j are periodic after some rank, and some are not. Without loss of generality, we assume $Y^1, \dots, Y^{k_1}$ are periodic, and $Y^{k_1+1}, \dots, Y^k$ are not. Almost certainly, each sequence $(Y^1)_{i \geq n}, \dots, (Y^{k_1})_{i \geq n}$ is periodic (n is big enough). Using $P \in \mathbb{N}$ to denote the smaller common multiple of those sequences' periods, we notice that $C_{n+Pi}(x^1) = C_n(x^1), \dots, C_{n+Pi}(x^{k_1}) = C_n(x^{k_1})$. We note $y_i = n + Pi$.

As the sequences $Y^{k_1+1}, \dots, Y^k$ are not eventually periodic, we know that each sequence $(f_n(x^{k_1+1}))_n, \dots, (f_n(x^k))_n$ is dense in $[0,1]$. Consequently, for any sequence of $k - k_1$ bits, there exists a countable number of $n^1 \in (Y_i)_{i \in \mathbb{N}}$ such that it is equal to $Y^{k_1+1}, \dots, Y^k$. In particular, if this sequence is $C_n(x^{k_1+1}), \dots, C_n(x^k)$, any of those n^1 different from n proves the theorem. □

In the case of the statistical goal learning problem in Sec. 2.2, we also have the same unlearnability. Consider the degenerate family of BR programs defined by the following BR program, and parametrized by $p \in [0,1]$:

```
**if** switch = True **then**
    $x \leftarrow f_n(p)$
    switch = False
**end if**
```

where switch is a Boolean variable which always have the initial value True, n is a natural number whose value is used by the BR program but cannot be known, and f_n is still the logistic map. This is a completely unpractical BR program as the output does not depend on the input variables at all. We obviously have $\forall x \in \mathbb{R}, P(p,x) = f_n(p)$. Supposing we use a tolerance $\varepsilon = 0.5$, the statistical goal learning problem in Eq. 2.2 is in this case:

$$\left.\begin{array}{rcl}\min\limits_{p} \|p - p^0\| & & \\ f_n(p) & \leq & 0.5,\end{array}\right\}$$

According to Subsec. 2.2.2, learning this problem with a PAC-algorithm amounts to finding a concept h satisfying Eq. 2.5. With this family of BR programs, we have:

$$\forall p \in [0,1], h(p) = 1 \Leftrightarrow f_n(p) \leq 0.5$$

which is obviously a member of $(C_n)_n$. As we have proved that $(C_n)_n$ is unlearnable, there are unlearnable statistical goal learning problems.

It must be noted that no practical application would ever try to learn this type of concept class. A key part of the proof is allowing the concept class to be infinite and indexed by a natural number, without bounding that index at all. This is unlikely to happen for computational reasons, the usual way to represent a natural number being with integer or long typed variables.

The existence of such extreme cases of unlearnability is nevertheless a thing to be careful of. It must be noted that none of the aforementioned computational difficulties are impossibilities, and that such unlearnable concepts are thus possible problems for BR programs, among other Turing-complete programming languages.

An agorithm able of answering our statistical BR learning problem is thus impossible to define in the general case. However, subsets of BR programs can be defined in such a way that they are both industrially relevant and computationally learnable. BR programs used in industry to automate decision making must terminate over every possible input. Moreover, they must terminate in a finite and even bounded time, so as to produce useful decisions. We thus decide to focus on BR programs for which the execution loop has a bounded number of iterations, as they are representative of industrial BR programs.

Chapter 4

A Mathematical Programming based algorithm

In this chapter, we provide a learning algorithm applicable to subsets of BR programs which are often used in industry. In Sec. 4.1, we define Iteration Bounded BR (IBBR) programs using our basic BR interpreter, and characterize them in terms of the standard interpreter as well as using a declarative constraint-based formalism. In Sec. 4.2, we describe a learning algorithm based on MP which transforms the statistical learning problem from Eq. 2.2 into a Mixed-Integer Programming (MIP) problem. In Sec. 4.3, we define Linear IBBR programs, or LIBBR programs, and characterize them both in terms of the basic interpreter and the standard interpreter. We further use a common form taken by industrial BR programs to confirm the relevance of using such a restrained subset of BR programs. We then transform the MIP problem from Fig. 4.3 into an equivalent Mixed-Integer Linear Programming (MILP) problem for LIBBR programs in Sec. 4.4, and discuss its theoretical complexity in Sec. 4.5.

4.1 General Iteration Bounded BR programs

We work on the subset of BR programs which is the set of *Iteration Bounded* BR programs, henceforth called IBBR, that is to say BR programs that have a higher bound on the number iteration of the execution loop their execution requires, for all possible inputs.

BR programs used in industry almost always have a strictly bounded number of rule executions, since the automated decisions BRs are used for must be taken in a fixed and limited time window. In this section, we define IBBR programs in terms of both the basic and the standard interpreter, and clarify the relationship between IBBR programs in those two programming languages, clarify the expressive power of IBBR programs, then use Constraint Programming to describe IBBR programs in a declarative form.

4.1.1 IBBR programs in two BR languages

We prove that IBBR programs, interpreted by a standard BR interpreter $\mathscr{I}_S$ from Sec. 2.1, are exactly equivalent to IBBR programs in our simplified BR interpreter $\mathscr{I}_0$.

Theorem 4.1. *The Iteration Bounded BR programs in $\mathscr{I}_S$ are exactly Iteration Bounded BR programs in $\mathscr{I}_0$.*

Proof. Let P be a BR program meant to be executed using $\mathscr{I}_S$ over inputs $x \in X$. We note its rules $\mathscr{R}_r$, indexed by $r \in \{1, \dots, \rho\}$ with $\rho \in \mathbb{N}$, and we suppose without loss of generality that the rules have the form:

```
if $T_r(x)$ then
    $x \leftarrow A_r(x)$
end if
```

As $\mathscr{I}_S$ includes partial priority, we suppose each rule has an additional priority value π_r such that a rule with lower π values is never selected if a rule with higher π value could be selected instead. As those are relative values, we suppose without loss of generality that $\pi_r \in \mathbb{N}$ and $0 < \pi_r \leq R$.

When executed using $\mathscr{I}_0$, the BR program P_0 found in Fig. 4.1 is equivalent to P, with $r \in \{0, \dots, R\}$ and the new inputs being $timer = 0$, $priorValidity = 0$, $p = R$, $needsReset = (\mathsf{False}, \dots, \mathsf{False})$, as well as $validityStart = (-1, \dots, -1)$. The value of $x \in X$ when executing P_0 is naturally the same as the input of P. This is true by definition of the Refraction, Priority and Recency conflict resolution strategy elements.

If there exists $n \in \mathbb{N}$ s.t. the execution of $P(x)$ executes less than n rules for any $x \in X$, then the execution of P_0 for any valid input executes at most these rules:

- R rule executions to initialize the $validityStart_r$ values
- n rule executions corresponding to the n rules executed in P
- $n \times R$ executions of the last rule (exploration of the priority levels)
- $n \times R \times n$ executions of the next to last rule (searching for the oldest valid rule)
- n rule executions to reset the value of $needsReset_r$ (once after each execution of a rule from P)
- n rule executions to reset the value of $validityStart_r$ (once after each execution of a rule from P)

Consequently, P_0 is iteration bounded, with an upper bound on the number of rule executions $3n + R + nR + n^2R$.

Conversely, a BR program meant to be executed in $\mathscr{I}_0$ is naturally equivalent to the same program with each rule repeated twice in $\mathscr{I}_S$, since repeating the

```
if $T_1(x) = \mathsf{False} \land needsReset_1$ then
    $needsReset_1 \leftarrow \mathsf{False}$
    $validityStart_1 \leftarrow -1$
end if
...
if $T_\rho(x) = \mathsf{False} \land needsReset_\rho$ then
    $needsReset_\rho \leftarrow \mathsf{False}$
    $validityStart_\rho \leftarrow -1$
end if
if $T_1(x) \land needsReset_1 = \mathsf{False} \land validityStart_1 = -1$ then
    $validityStart_1 \leftarrow timer$
end if
...
if $T_\rho(x) \land needsReset_\rho = \mathsf{False} \land validityStart_\rho = -1$ then
    $validityStart_\rho \leftarrow timer$
end if
if $T_1(x) \land needsReset_1 = \mathsf{False} \land p = \pi_1 \land validityStart_1 = priorValidity$ then
    $x \leftarrow A_1(x);$              $p \leftarrow \rho$
    $timer \leftarrow timer + 1;$        $priorValidity \leftarrow 0$
    $needsReset_1 \leftarrow \mathsf{True}$
end if
...
if $T_\rho(x) \land needsReset_\rho = \mathsf{False} \land p = \pi_\rho \land validityStart_\rho = priorValidity$ then
    $x \leftarrow A_\rho(x);$              $p \leftarrow \rho$
    $timer \leftarrow timer + 1;$        $priorValidity \leftarrow 0$
    $needsReset_\rho \leftarrow \mathsf{True}$
end if
if $priorValidity \leq timer$ then
    $priorValidity \leftarrow priorValidity + 1$
end if
if $p > 0$ then
    $p \leftarrow p - 1$
    $priorValidity \leftarrow 0$
end if
```

Figure 4.1: **BR program equivalent to the Iteration Bounded program** P when executed using $\mathscr{I}_0$, with inputs being:
$(x \in X$, $timer = 0$, $priorValidity = 0$, $p = R$,
$needsReset = (\mathsf{False}, \dots, \mathsf{False})$, $validityStart = (-1, \dots, -1))$

same rule twice negates the refraction element, with each rule having a distinct priority level corresponding to its position in the original formulation. Given a BR program meant to be executed in $\mathscr{I}_0$ which executes at most $n \in \mathbb{N}$ rules for each of its inputs, this equivalent program in $\mathscr{I}_S$ also executes at most n rules. IBBR programs in $\mathscr{I}_0$ are thus equivalent to iteration bounded programs in $\mathscr{I}_S$. □

4.1.2 Complexity of IBBR programs

We now explore the expressive power of IBBR programs. Firstly, it must be noted that without the restriction over the complexity of conditions and actions of BRs, IBBR programs remain Turing-complete, as we can use a Turing-undecidable function as part of the action of a rule. For example, suppose $f(x, T)$ is a function modeling the halting problem: given an input x and a Turing machine T, it returns 0 if T halts on input x and 0 otherwise. We recall the BR program described at the beginning of Chapter 3, and use it to simulate f. The resulting BR program takes as input a value x, a Turing machine T and an initially True boolean switch, and contains the single following rule:

```
if switch = True then
    $x \leftarrow f(x, T)$
    switch ← False
end if
```

This BR program is both an IBBR and undecidable, as it executes at most one rule yet is equivalent to the halting problem, which is undecidable [126]. However, it does not respect the restrictions we use over the complexity of actions in BRs, as the action $x \leftarrow f(x, T)$ is as complex as the halting problem.

We now suppose again that all BR have a condition T and an action A which can be computed in polynomial time.

Theorem 4.2. *Let τ be an upper bound on the computational complexity of any* *one condition or action of a BR in the BR program $P(p, .)$. The complexity of executing $P(p, x)$ with the basic interpreter for any $x \in X$ is $O(\tau\, n\, \rho)$, where n is the upper bound on the number of iterations of the execution loop and ρ is the number of BRs in the BR program.*

Proof. The proof is a direct consequence of the structure of the execution loop as we described it in Sec. 2.1.2. The complexity of the execution is at most the complexity of executing ρ tests of complexity τ and an action of complexity τ during each loop, of which there are at most n by definition of IBBR programs. The maximal complexity is thus $n\,\tau\,(\rho + 1)$. □

4.1.3 Declarative form of an IBBR programs

When a BR program has a bounded number of iterations of the execution loop, we can write a Constraint Satisfaction Problem such that its resolution is equivalent to the execution of the BR program. This is done by explicitly naming the values of x after each rule execution. When this Constraint Satisfaction Problem is easy to solve, the statistical goal learning problem can also be solved by simply adding a few constraints and an objective function, transforming the learning problem into a Mathematical Optimization problem.

Let us consider an iteration bounded BR program made up of ρ rules $\mathcal{R}_r$, which we assume without loss of generality are of the form:

```
**if** $T_r(x)$ **then**
    $x \leftarrow A_r(x)$
**end if**
```

We exhibit in Fig. 4.2 a set of constraints modeling the execution of the BR program for an input $q \in X$. The iterations of the execution loop are indexed by $i \in I = \{1, \dots, n\}$ where $n - 1$ is the upper bound on the number of iterations, the final value of x corresponds to iteration n. The rules are indexed by $r \in R = \{1, \dots, \rho\}$. We use an auxiliary binary variable $y_{i,r}$ with the property: $y_{i,r} = 1$ iff the rule $\mathcal{R}_r$ is executed at iteration i. We conflate the boolean and binary truth values: $\mathsf{True} = 1$ and $\mathsf{False} = 0$.

We note (C1), (C2), etc. the constraints related to the evolution of the execution and (IC1), (IC2), etc. the constraints related to the initial conditions of the BR program:

- (C1) represents the evolution of the value of the variable x
- (C2) represents the property that at most one rule is executed per iteration
- (C3) represents the fact that a rule whose condition is False cannot be executed
- (C4) represents the fact that the first rule whose condition is True must be executed
- (IC1) represents the initial value of x

$$\begin{array}{lll}
\forall i \in I \backslash \{n\} & x^{i+1} = \sum_{r \in R} A_r(x^i) y_{i,r} + (1 - \sum_{r \in R} y_{i,r}) x^i & \text{(C1)} \\
\forall i \in I & \sum_{r \in R} y_{i,r} \leq 1 & \text{(C2)} \\
\forall (i,r) \in I \times R & y_{i,r} \leq T_r(x^i) & \text{(C3)} \\
\forall (i,r) \in I \times R & y_{i,r} \geq T_r(x^i) - \sum_{r' < r} y_{i,r'} & \text{(C4)} \\
 & x^1 = q & \text{(IC1)} \\
\forall i \in I & x^i \in X & \\
\forall (i,r) \in I \times R & y_{i,r}, \in \{0,1\} &
\end{array}$$

Figure 4.2: **Set of Constraints Modeling the Execution of a BR Program**

Theorem 4.3. *The constraints from Fig. 4.2 correctly model the execution of an IBBR program s.t. $n-1$ is the upper bound on the number of iterations, with input q. The value of x^n after applying the constraints is then the output of the BR program.*

Proof. We begin by proving that for a given $i \in I$, it is true that $y_{i,r} = 1$ iff x^i fulfills the condition for rule $\mathcal{R}_r$ and does not fulfill the condition for any rule $\mathcal{R}_{r'}$ where $r' < r$. Suppose $y_{i,r} = 1$. (C3) $\Rightarrow T_r(x^i) = 1 = \mathsf{True}$, meaning that x^i fulfills the condition for rule $\mathcal{R}_r$. Let us now set $r' < r$.

$$\text{(C2)} \Rightarrow y_{i,r'} = 0 \wedge \sum_{r''<r'} y_{i,r''} = 0$$
$$\text{(C4)} \Rightarrow y_{i,r'} \geq T_{r'}(x^i) - \sum_{r''<r'} y_{i,r''}$$

This in turn gives us $T_{r'}(x^i) = 0 = \mathsf{False}$, meaning that x^i does not fulfill the condition for rule $\mathcal{R}_{r'}$.

Conversely, suppose that x^i fulfills the condition for rule $\mathcal{R}_r$ and does not fulfill the condition for any rule $\mathcal{R}_{r'}$ where $r' < r$. Reasoning by induction over r', we see that assuming $\sum_{r''<r'} y_{i,r''} = 0$ (which is true for $r' = 1$) we have:

$$C4 \Rightarrow y_{i,r'} = 0$$

because the condition for $\mathcal{R}_{r'}$ is not fulfilled. We thus have $\sum_{r'<r} y_{i,r'} = 0$. This and the fact that the condition for $\mathcal{R}_r$ is fulfilled means that $y_{i,r} = 1$.

A simple inductive proof over the $i \in I$ then proves that the x^i are the successive values taken by x during the execution of the BR program as long as $\sum_{r\in R} y_{i,r} = 1$ and that the value of x^i does not change as long as $\sum_{r\in R} y_{i,r} = 0$, which corresponds to the stopped execution of the BR program. This also proves that the final value x^n is the output of the BR program. □

4.2 A learning algorithm: MP

Learning the parameters that solve the statistical goal learning problem from Eq. 2.2 in such a BR program can then be achieved by solving a derived Mathematical Optimization problem. In this section, we describe an MP problem which is exactly equivalent to Eq. 2.2 for IBBR programs.

Let us assume that a family of BR programs indexed by p is iteration bounded. We wish to modify a given program indexed by p^0. We write each BR program $P(p,.)$ and assume that the execution of $P(p,q)$ iterates the execution loop strictly less than $n \in \mathbb{N}$ times, for any $q \in X$. In other words, $n-1$ is the upper bound on the number of iterations. We index the instances in the set of known inputs Q with $j \in J = \{1,\ldots,m\}$, where $m = \text{card}(Q)$ is the number of instances in the training set. The parameter p is now one of the variables.

Every constraint numbered as before fulfills the same role, adding the indexation j, with the simple change that A_r and T_r become $A_{p,r}$ and $T_{p,r}$. The additional constraints are:

- (C5) represents the need for the computation to have terminated after $n-1$ executions
- (C6) represents the goal from Eq. 2.2, that is a constraint over the average of the final values of x.

Theorem 4.4. *The MP problem from Fig. 4.3 finds a value of p that satisfies Eq. 2.2.*

Proof. The proof derives directly from Th. 4.3. As we reuse all the constraints from Fig. 4.2, we know that $x^{n,j} = P(p, x^{1,j})$. Furthermore, let p^* be an optimal

$$\boxed{\begin{array}{lll}
\underset{p,x,y}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (C1), (C2),(C3), (C4), (IC1) & \\
\quad\forall j \in J & \sum_{r\in R} y_{n,j,r} = 0 & (C5) \\
 & \left|\frac{1}{m}\sum_{j\in J} f(x^{n,j}) - g\right| \leq \varepsilon & (C6) \\
\quad\forall (i,j) \in I{\times}J & x^{i,j} \in X & \\
 & p \in \mathbb{R} & \\
\quad\forall (i,j,r) \in I{\times}J{\times}R & y_{i,j,r}, \in \{0,1\} &
\end{array}}$$

Figure 4.3: **Mathematical Optimization Problem Formulation for Eq. 2.2**

solution to the MP problem in Fig. 4.3. From constraint (C6), we know that:

$$\left|\frac{1}{m}\sum_{j\in J} f(x^{n,j}) - g\right| \leq \varepsilon$$

This is directly equivalent to:

$$\left|\,\mathbb{E}_{q\in Q}\Big[f(P(p,q))\Big] - g\,\right| \leq \varepsilon$$

Consequently, p^* is a member of the following set S:

$$S = \{p \in \pi \mid \left|\,\mathbb{E}_{q\in Q}\Big[f(P(p,q))\Big] - g\,\right| \leq \varepsilon\}$$

To satisfy Eq. 2.2, p^* now needs to be a lower bound of that set.

Let $p \in S$. From Th. 4.3 and the fact that all constraint of Fig. 4.2 are in

Fig. 4.3, we can say:

$$\forall q \in Q, j \in J, x^{1,j} = q \Rightarrow x^{n,j} = P(p, x^{1,j})$$

With that information, $p \in S$ gives us:

$$\left| \frac{1}{m} \sum_{j \in J} f(x^{n,j}) - g \right| \leq \varepsilon$$

which is exactly (C6). The proof of Th. 4.3 also proves that the $y_{i,j,r}$ have the property: for a given $i \in I$, it is true that $y_{i,j,r} = 1$ iff $x^{i,j}$ fulfills the condition for rule $\mathcal{R}_r$ and does not fulfill the condition for any rule $\mathcal{R}_{r'}$ where $r' < r$. As the family of BR programs P is assumed in this Section to be Iteration Bounded with at most $n - 1$ iterations, we know that $x^{n,j}$ does not fulfill the condition for any rule. Consequently,

$$\forall j \in J, r \in R, y_{n,j,r} = 0$$

which gives us (C6). Th. 4.3 already gives us (C1) through (C4). We now know that p is a feasible solution of the MP problem in Fig. 4.3. By the definition of an optimal solution, we thus have:

$$\forall p \in S, p^* \leq p$$

We have proved that p^* is a minimum of set S, which defines exactly a solution of Eq. 2.2. □

As modifying the parameter means modifying the BR program, solving this mathematical optimization problem is predicated on having entire families of BRs which are proven to be iteration bounded. In practice however, industrial applications produce families of BR programs which are not explicitly iteration bounded: instead, the BR program using the existing parameter value p_0 is iteration bounded, and the desired BR program with the 'correct' parameter value p must be iteration bounded, with a numerical bound on the number of iterations per execution. In such cases, when the solution needs to be iteration bounded, proving that the whole family of BR programs is iteration bounded is not necessary, as this formulation only finds solutions which terminate in less than $n-1$ iterations. Our algorithm is thus useful even in cases where the family of BR programs in not known to be iteration-bounded, provided that the original BR program on which the family is based (in the sense of Def. 2.2) is itself iteration-bounded.

The feasibility of using such a method to solve Eq. 2.2 is somewhat difficult to evaluate, as it strongly depends on the form of the BRs we wish to learn, i.e. the form of the *conditions* and *actions*. In particular, Linear BR programs have an easily linearizable MP form which we explore in the next chapter. Some IBBR programs are still unlearnable, of course, since IBBR programs in the most general sense are still Turing-complete, as seen in Subsec. 4.1.2. Even when only considering polynomial conditions and actions, however, it seems very difficult to scale such a method for a greater number of rules ρ, training input values m, or iterations n as the number of variables and constraints is directly correlated to those values.

While industrial applications often have a limited value of n ($n \leq 100$), the same cannot be said about the number of rules: some BR programs have over 10,000 rules! The value of m can be adjusted according to the difficulty of the mathematical program, but of course the quality of the learning is better the more training data is included.

4.3 Linear Iteration Bounded BR programs

We now seek to refine our subset of BR programs further in order to make the MP based algorithm from the previous chapter into an algorithm which can be used automatically using standard MP solvers. In particular, we are interested in linearity properties of the MIP problem in Fig. 4.3.

In this section, we define Linear IBBR programs, or LIBBR programs, and characterize them both in terms of the basic interpreter and the standard interpreter. We prove that Decision Trees in BR form, as are often created by BRMS users, are LIBBR programs.

4.3.1 Linear BR programs

We define Linear BR programs as programs with input in $X \subset \mathbb{R}^d$ which can be written entirely using Linear BRs, i.e. BRs of the form:

```
if $L \leq x \leq H$ then
    $x \leftarrow Ax + B$
end if
```

with $L, H, B \in \mathbb{R}^d$ and $A \in \mathbb{R}^{d \times d}$. Note that while this form does not have meta-variables, BR programs written entirely with rules of the form:

```
if $L^\alpha \leq \alpha \leq H^\alpha \wedge L \leq x \leq H$ then
    $\alpha \leftarrow A^\alpha \alpha + B^\alpha$
    $x \leftarrow Ax + B$
end if
```

are also linear BR programs, since they can be rewritten by simply re-using the set of elementary rules as a set of rules.

While many industrial BR programs use complex functions, such as scoring functions in loan validation programs, many simpler decision making processes are also often automated using BRs. Any BR program which simply applies a decision table, for example, can be written as a linear BR program.

The computational complexity of LIBBR programs is linear in both n and ρ. In the next theorem, we assume that the complexity of a scalar comparison, a scalar multiplication and a scalar addition are all 1.

Theorem 4.5. *The computational complexity C of a LIBBR program $P(p,.)$ with ρ rules and inputs in $X \subseteq \mathbb{R}^d$ when executed using our basic interpreter is:*

$$C \leq 2\,n\,d\,(\rho + d)$$

where n is the upper bound on the number of iterations of the iteration loop.

Proof. We prove this theorem using a direct application of the execution loop. As $P(p,.)$ is a LIBBR program, it iterates at most n times. Each iteration has complexity at most the sum of:

- $2\rho\,d$ (two comparisons for each evaluated condition)
- $d\,(2\,d - 1)$ (the naive matrix multiplication $A\,x$)
- d (the addition of B)

Doing the sum of those terms, we indeed find that the upper bound on the complexity is n times $2\,d\,(\rho + d)$. □

A better performance can be obtained by storing the results of the tests performed in the condition of the rules, and only re-evaluating the conditions involving the components of x which have been modified by the previous assignment action – this is the basis of the Rete algorithm [43]. The complexity gain by such an algorithm relies on assumptions of sparseness in the actions (in particular of sparseness in the matrix A of each rule). We refrain from any such assumption and only note that in all industrial cases, $d \ll \rho$: we consider the complexity of executing a LIBBR program to be $O(n\,d\,\rho)$.

We now prove that linear BR programs, interpreted by a standard BR interpreter $\mathscr{I}_S$, are exactly equivalent to linear BR programs in our simplified BR interpreter $\mathscr{I}_0$.

Theorem 4.6. *Linear Iteration Bounded BR programs in $\mathscr{I}_S$ are exactly Linear Iteration Bounded BR programs in $\mathscr{I}_0$.*

Proof. The idea stems from the remark that the BR program in Fig. 4.1 has almost only linear additions compared to the initial BR program as written for interpreter $\mathscr{I}_0$. We use the same pattern as in the proof of Th. 4.1.

Given a LIBBR program P meant to be executed by $\mathscr{I}_S$, with rules indexed by $r \in \{1, \dots, \rho\}$ and n the maximum number of iterations of the execution loop $P(x)$ for any $x \in X$, the only non-linear parts of the BR program P_0 described in Fig. 4.1 are the conditions:

- $validityStart_r = priorValidity$
- $priorValidity \leq timer$

The first item is an equality condition between bounded integers and can thus be linearized by enumerating couples of integers in $\{0, \dots, n\}$. The second item can be linearized in the same way using the conditions $priorValidity \leq k \wedge timer = k$ for each $k \in \{0, \dots, n\}$. The BR program P_0 can be transformed into a linear program and is also iteration bounded as shown in the proof of Th. 4.1. As the two transformations do not add any iteration to the execution loop in $\mathscr{I}_0$ (the number of actions executed stays the same), P can be expressed as a LIBBR in $\mathscr{I}_0$.

Conversely, the same transformation used in the proof of Th. 4.1 to prove that an IBBR program P_0 in $\mathscr{I}$ can be expressed as an IBBR P_S in $\mathscr{I}_S$ proves the same for LIBBR programs. Repeating each BR of a P_0 twice does not change the form of the BRs, so P_S is still Linear. As it is also iteration bounded from the proof of Th. 4.1, P_S is a LIBBR in $\mathscr{I}_S$ which is equivalent to P_0 in $\mathscr{I}$. $\square$

4.3.2 Example: Decision Trees

A specific application of studying iteration bounded BR program is found in a commonly used form of decision-making process: decision trees. Decision trees are easily learned from scratch using supervised learning algorithms found in software such as IBM SPSS Modeler [117] or R [103]. The most famous such algorithms are C4.5 [23] and its predecessor ID3 [56]. Such decision trees can then be modeled in BRMS, whether automatically (as can be done using IBM SPSS Modeler and IBM ODM [118]) or manually.

All decision trees can be modeled directly as an iteration bounded Linear BR program. We use the definition of decision trees given by Quinlan in [124].

Definition 4.1. A decision tree is either a leaf assigning a constant value to a variable x_0, or a structure of the form:

```
if $C_1$ then
    $D_1$
else if $C_2$ then
    $D_2$
    ...
else if $C_n$ then
    $D_n$
end if
```

such that the C_i are mutually exclusive conditions and the D_i are themselves decision trees, where the conditions all have the form $x_k < t$ or $x_k > t$ if x_k is continuous; or $x_k \in V \subset X_k$ if x_k is discrete with values in X_k.

Any decision tree can be written as a set of production rules (i.e. BRs) [124]. Furthermore, detailed reading of Quinlan's article provides the proof that for decision trees of the provided form, the BRs are indeed Linear. We prove that these

BR programs are also iteration bounded, and are thus LIBBR programs.

Theorem 4.7. *Let D be a Decision Tree. Then D can be written as an LIBBR program.*

Proof. We prove our theorem by inductively constructing a BR program which simulates D. Note that this does not produce an efficient rewrite as in [124] or [102]. We suppose that the vector of variables x is indexed from 1 to K and use x_0 to encode the output class given by the Decision Tree.

Suppose D is already a leaf of a decision tree, i.e. it simply states an output class y. It is simulated by a BR program using a single BR in $\mathscr{I}_S$, which is obviously iteration bounded and Linear:

```
if True then
    $x_0 \leftarrow y$
end if
```

We now suppose that a decision tree of depth d can be simulated by an iteration bounded Linear BR program. Suppose D has a depth of d with $d > 1$, it has the form:

$$(\text{if } T_1(x) \text{ then } D_1, \ldots, \text{ if } T_n(x) \text{ then } D_n)$$

where the $T_k(x)$ are mutually exclusive tests. We note the BRs encoding the iteration bounded Linear BR program which simulates D_k as rules indexed by i:

```
if $T_{i,k}(x)$ then
    $x_0 \leftarrow y_{i,k}$
end if
```

If the T_k are tests over a continuous variable, then the form prescribed by the theorem is linear and the following Linear BR program simulates D:

```
if $T_k(x) \wedge T_{i,k}(x)$ then
    $x_0 \leftarrow y_{i,k}$
```

```
**end if**
```

If the T_k are tests over a discrete variable, then noting each $T_k(x)$ as $x_t \in V_k$ and indexing V_k over j, the following Linear BR program simulates D:

```
if $v_{j,k} \leq x_t \leq v_{j,k} \wedge T_{i,k}(x)$ then
    $x_0 \leftarrow y_{i,k}$
end if
```

Furthermore, as the T_k are mutually exclusive, execution of either of these BR program on a given input x has the same number of iterations of the execution loop as the execution of the BR program simulating one of the D_k. As each of those is iteration bounded, this BR program is also iteration bounded, with the bound being the maximum of the bounds for the programs of each sub-tree. $\square$

Users of decisions trees will note that this theorem does not hold for some accepted variants of decision trees. Two popular extensions are tests including more than one variable, which can lead to non-linear conditions; and branching incorporating actions between two condition nodes of the decision tree, which can lead to loss of the iteration bounded property.

4.4 An algorithm for LIBBR programs

In this section and in Chapter 5, we use reformulations of various constraints we introduced in Sec. 4.2. In such cases, we suffix 'primes' (or other qualifiers) to the constraint names. For example, (C1) was nonlinear in Fig. 4.2 because of the unspecified function $A_r(x^i)$; in Fig. 4.4 the function $A(\cdot)$ is instantiated in (C1$'$), and in Fig. 4.5 the remaining products are linearized exactly in (C1$''_1$), (C1$''_2$) and (C1$''_3$). If a constraint needs to be split, its name is indexed by an ordinal, e.g. (C4) becomes (C4$_1$), (C4$_2$) and (C4$_3$) in Fig. 4.4.

In the rest of this chapter, we study an LIBBR program $P(p, q)$ with inputs $q \in X \subset \mathbb{R}^d$, $d \in \mathbb{N}$, and with a rule set $\{\mathcal{R}_r \mid r \leq \rho\}$ containing rules of the form:

```
**if** $L_r \leq x \leq H_r$ **then**
    $x \leftarrow A_r x + B_r$
**end if**
```

with rule $\mathcal{R}_1$ being instead:

```
**if** $L_1 \leq x \leq H_1$ **then**
    $x \leftarrow A_1^p x + B_1$
**end if**
```

where A_1^p is a $d \times d$ matrix satisfying:

$$\begin{cases} \forall h, k \in D, h \neq 1 \vee k \neq 1 \Rightarrow (A_1^p)_{h,k} & = & (A_1)_{h,k} \\ (A_1^p)_{1,1} & = & p \end{cases}$$

with $D = \{1, \ldots, d\}$. We also assume that X is bounded, which given that the BR program is bounded imposes the existence of a big M, an upper bound on the absolute value of any scalar appearing during the execution of $P(p, q)$. Furthermore, we assume that the statistical goal learning problem we wish to solve considers the relevant output to be x_1, i.e. $f(P(p, q)) = P(p, q)_1$.

In the rest of this thesis, we concatenate indices so that $(L_r)_k = L_{rk}$, $(G_r)_k = G_{rk}$, $(A_r)_{hk} = A_{rhk}$ and $(B_r)_k = B_{rk}$. We assume that rules are meaningful, such that $L_k \leq G_k$.

In this section, we first transform the MIP problem from Fig. 4.3 for the case where Eq. 2.2 learns the straight average of a single output, using the output function $f(P(p, q)) = P(p, q)_1$, in the LIBBR case. The resulting MIP problem is linearizable, and we prove the equivalence to a MILP formulation in Subsec. 4.4.2. Finally, we evaluate the applicability to our algorithm to industrial problems by estimating its theoretical complexity.

4.4.1 A learning algorithm: MIP

The Constraint Satisfaction Problem from Subsec. 4.1.3 can be written as the set of MIP constraints from Fig. 4.3 and still model the execution of the BR program. In turn, this means the MP formulation from Fig. 4.3 can be transformed into the MIP problem described in Fig. 4.4.

Like before, the iterations of the execution loop are indexed by $i \in I = \{1, \dots, n\}$ where $n-1$ is the upper bound on the number of iterations, the final value of x corresponds to iteration n. The rules are indexed by $r \in R = \{1, \dots, \rho\}$. The binary variable y_{ijr} still has the property: $y_{ir} = 1$ iff the rule $\mathcal{R}_r$ is executed at iteration i in the execution of the program on the input j. We now use the vectors of binary variables y^H_{ijr} and y^L_{ijr} to enforce this property. In the problem we examine, the parameter is assumed to take the place of $A_{1,1,1}$, so we note a an additional variable initialized to $a = A$ except for $a_{1,1,1} = p$. We assume that a value for the big M is known, it could for example be $M \geq 2\sup\limits_{x \in X}(|x|)$. Similar sets of constraints exists for when the parameter p takes the place of a scalar in B_r, H_r or G_r.

Every constraint numbered as before fulfills the same role. The additional constraints are:

- (C1′) still represents the evolution of the value of the variable x
- (C3′) still represents the fact that a rule whose condition is False cannot be executed
- (C4′$_1$) through (C4′$_3$) represent the fact that the first rule whose condition is True must be executed

- (C6′) still represents the goal from Eq. 2.2
- (IC2) through (IC4) represent the initial value of A

$$
\begin{array}{llr}
\underset{p,a,x,y,y^H,y^L}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (C2), (C5), (IC1) & \\
\forall (i,j) \in I\backslash\{n\}\times J & x^{i+1,j} = \sum_{r\in R}(a_r x^{i,j} + B_r) y_{ijr} + (1 - \sum_{r\in R} y_{ijr}) x^{i,j} & (C1') \\
\forall (i,j,r) \in I\times J\times R & L_r - M(1 - y_{ijr})e \le x^{i,j} \le H_r + M(1 - y_{ijr})e & (C3') \\
\forall (i,j,r,k) \in I\times J\times R\times D & x^{i,j}_k \ge H_{rk} - M y^H_{ijrk} - M \sum_{r'<r} y_{ijr'} & (C4'_1) \\
\forall (i,j,r,k) \in I\times J\times R\times D & x^{i,j}_k \le L_{rk} + M y^L_{ijrk} + M \sum_{r'<r} y_{ijr'} & (C4'_2) \\
\forall (i,j,r) \in I\times J\times R & 2d - 1 + y_{ijr} \ge \sum_{k\in D} (y^H_{ijrk} + y^L_{ijrk}) & (C4'_3) \\
 & \left| \frac{1}{m} \sum_{j\in J} x^{n,j}_1 - g \right| \le \varepsilon & (C6') \\
\forall r \in \{2, \dots, \rho\}, & a_r = A_r & (IC2) \\
 & a_{1,1,1} = p & (IC3) \\
\forall (h,k) \in D^2 \setminus \{1,1\}, & a_{1hk} = A_{1hk} & (IC4) \\
\forall (i,j) \in I\times J & x^{i,j} \in X & \\
\forall k \in R & a_k \in \mathbb{R}^{d\times d} & \\
 & p \in \mathbb{R} & \\
\forall (i,j,r,k) \in I\times J\times R\times D & y_{ijr}, y^H_{ijrk}, y^L_{ijrk} \in \{0,1\} &
\end{array}
$$

Figure 4.4: **MIP Formulation for Solving Eq. 2.2** with $e = (1, \dots, 1) \in \mathbb{R}^d$ a vector of all ones

Theorem 4.8. *The MIP problem from Fig. 4.4 finds a value of p that satisfies Eq. 2.2.*

The proof derives directly from Th. 4.4 and is expanded in App. A.1. Solving this MIP problem using standard Branch-and-Bound algorithms is always possible, but it can become expensive very fast as n, m and ρ increase.

4.4.2 A learning algorithm: MILP

Careful examination of the MIP problem in Fig. 4.4 reveals that it is almost linearizable. In fact, a Mixed-Integer Linear Programming (MILP) formulation of the problem is always possible for iteration bounded, linear BR program given a simple restriction:

$$\forall r \in R, (h,k) \in K^2, A_{rhk} = p \Rightarrow A_{rhk} \in \{0,1\} \tag{4.1}$$

As MILP problems are much easier to solve than general MIP problems, this is a simple step to take. Furthermore, this restriction does not always apply if x is a mixed vector of continuous and integer variables. Supposing $k \in K$ such that $x_1, \ldots, x_k$ are discrete variables and $x_{k+1}, \ldots, x_d$ are continuous variables, the restriction on A becomes:

$$\forall r \in R, A_r = \begin{pmatrix} A_r^1 & 0 \\ A_r^2 & A_r^3 \end{pmatrix}$$

with $A_r^1 \in \mathbb{N}^{k \times k}$, $A_r^2 \in \mathbb{R}^{(d-k) \times k}$ and A_r^3 follows the condition in Eq. 4.1.

A straightforward linear reformulation of Fig. 4.4 exists for when the parameter p takes the place of a scalar in B_r, L_r or G_r. Fig. 4.5 describes such a MILP problem when p takes the place of $B_{1,1}$. We linearize the products of $A_r x^{i,j} + b_r$ by y_{ijr} and $x^{i,j}$ by y_{ijr} in (C1) using factorization and an auxiliary variable $w \in \mathbb{R}^{I \times J \times R}$. We arrange to have $w_{ijr} = (A_r x^{i,j} + b_r - x^{i,j}) y_{ijr}$, i.e. $w_{ijr} = A_r x^{i,j} + b_r - x^{i,j}$ (the difference between the new and the old values of x^j) iff rule r is executed, and 0 otherwise. The value of M should increase accordingly.

Theorem 4.9. *The MILP problem in Fig. 4.5 finds a value of p that satisfies Eq. 2.2, when p takes the place of B_{11}. A similar MILP problem exists for when p takes the place of another scalar in B_{rk}, L_{rk} and G_{rk}.*

$$\begin{array}{lcr}
\underset{p,b,x,y,y^H,y^L,w}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\text{C2}), (\text{C3}'), (\text{C4}'_1), (\text{C4}'_2), (\text{C4}'_3), (\text{C5}), (\text{IC1}) & \\
\forall (i,j) \in I \backslash \{n\} \times J & x^{i+1,j} = \sum_{r \in R} w_{ijr} + x^{i,j} & (\text{C1}''_1) \\
\forall (i,j) \in I \times J \times R & -M y_{ijr} e \le w_{ijr} \le M y_{ijr} e & (\text{C1}''_2) \\
\forall (i,j,r) \in I \times J \times R & A_r x^{i,j} + b_r - x^{i,j} - M(1 - y_{ijr}) e & \\
 & \le w_{ijr} \le A_r x^{i,j} + b_r - x^{i,j} + M(1 - y_{ijr}) e & (\text{C1}''_3) \\
 & -\varepsilon \le \frac{1}{m} \sum_{j \in J} x_1^{n,j} - g \le \varepsilon & (\text{C6}'') \\
\forall r \in \{2, \dots, \rho\} & b_r = B_r & (\text{IC2}') \\
 & b_{1,1} = p & (\text{IC3}') \\
\forall k \in \{2, \dots, d\} & b_{1k} = B_{1k} & (\text{IC4}') \\
\forall (i,j) \in I \times J & x^{i,j} \in X & \\
\forall (i,j,r) \in I \times J \times R & b_r, w_{ijr} \in \mathbb{R}^d & \\
 & p \in \mathbb{R} & \\
\forall (i,j,r,k) \in I \times J \times R \times D & y_{ijr}, y^H_{ijrk}, y^L_{ijrk} \in \{0,1\} &
\end{array}$$

Figure 4.5: **MILP Formulation with p Taking the Place of B_{11}** with $e = (1, \dots, 1) \in \mathbb{R}^d$ a vector of all ones

The proof derives directly from Th. 4.8, by factoring constraint (C1) in Fig. 4.4 and studying the possible values of y_{ijk}. It is expanded in App. A.2.

When the parameter takes the place of $A_{1,1,1}$, the linear formulation is only possible if $A_{1,1,1}$ is a discrete variable. For readability purposes, we assume thereafter that p only takes the place of a component in $A_r, r \in R$ when we have $\forall r \in R, A \in \{0,1\}^{d \times d}$. The associated MILP problem is in Fig. 4.6. In that case, we have the additional product of ax to linearize, so we use another auxiliary variable $z \in \mathbb{R}^{I \times J \times R \times D^2}$ such that $z_{ijrhk} = a_{rhk} x_k^{i,j}$. Again, the value of M increases in most cases.

Theorem 4.10. *The MILP problem in Fig. 4.6 finds a value of p that satisfies Eq. 2.2, when p takes the place of A_{111} and A_{1hk} are binary variables.*

The proof derives from Th. 4.9 and a study of the possible values of A_{1hk} and

$$
\begin{array}{lll}
\underset{p,a,x,y,y^H,y^L,w,z}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\mathrm{C1}''_1), (\mathrm{C1}''_2), (\mathrm{C2}), (\mathrm{C3}'), (\mathrm{C4}'_1), (\mathrm{C4}'_2), (\mathrm{C4}'_3) & \\
 & (\mathrm{C5}), (\mathrm{C6}''), (\mathrm{IC1}), (\mathrm{IC2}), (\mathrm{IC3}), (\mathrm{IC4}) & \\
\forall (i,j,r,h) \in I{\times}J{\times}R{\times}D & \sum_{k\in D} z_{ijrhk} + B_{rh} - x^{i,j}_h - M(1 - y_{ijr}) & \\
 & \leq w_{ijrh} \leq \sum_{k\in D} z_{ijrhk} + B_{rh} & (\mathrm{C1}'''_3) \\
 & - x^{i,j} + M(1 - y_{ijr}) & \\
\forall (i,j,r) \in I{\times}J{\times}R & -Ma_r \leq z_{ijr} \leq Ma_r & (\mathrm{C1}'''_4) \\
\forall (i,j,r,h,k) \in I{\times}J{\times}R{\times}D^2 & x^{i,j}_k - M(1 - a_{rhk}) & \\
 & \leq z_{ijrhk} \leq x^{i,j}_k & (\mathrm{C1}'''_5) \\
\forall (i,j,r,h,k) \in I{\times}J{\times}R{\times}D^2 & x^{i,j}, z_{ijrhk} \in X & \\
\forall (i,j,r) \in I{\times}J{\times}R & w_{ijr} \in \mathbb{R}^d & \\
\forall r \in R & a_r \in \{0,1\}^{d\times d} & \\
 & p \in \{0,1\} & \\
\forall (i,j,r,k) \in I{\times}J{\times}R{\times}D & y_{ijr}, y^H_{ijrk}, y^L_{ijrk} \in \{0,1\} &
\end{array}
$$

Figure 4.6: **MILP Formulation with p Taking the Place of A_{111}** with $e = (1,\ldots,1) \in \mathbb{R}^d$ a vector of all ones

y_{ijr}. It is expanded in App. A.3.

We can trivially expand the MILP problem to optimize over more than one parameter, adding constraints similar to constraints (IC2), (IC3) and (IC4) or (IC2′), (IC3′) and (IC4′) in Fig. 4.4 or Fig. 4.9 as necessary and having an objective of $\sum_p \|p^0 - p\|$.

The feasibility of our approach for such a class of BR programs is examined in the next part. Note that, as in Sec. 4.2, the iteration bounded property of the BR program is an intrinsic part of the mathematical program. As such, even BR programs which were not originally iteration bounded (or which did not stop after only n iterations) can be learned this way, although the existence of a feasible solution is much less certain than otherwise.

4.5 Theoretical complexity

How useful is the algorithm defined in Sec. 4.2 and Sec. 4.4? The answer depends on the complexity of the Optimization Problem derived from the BR program. The complexity of parsing an iteration bounded linear BR program to obtain an Optimization Problem is relatively small, being $O(\rho)$, where ρ is the number of rules in the BR program.

To evaluate the rest of the algorithm, we first analyze the descriptive complexity of the MILP problem for an iteration bounded linear BR program. We then provide the complexity of solving the MILP problem in the worst case using an off-the-shelf solver such as CPLEX, and estimate how much improvement might be obtained by using custom heuristics for the task.

Let P be an iteration bounded linear BR program. We assume that P has ρ rules. Its inputs $x \in X \subseteq \mathbb{R}^d$ are vectors of d variables. n is the maximum number of times P executes a rule. The learning problem defined in Eq. 2.2 is defined by p a vector of ϕ parameters and a testing set Q of cardinality m. In the MILP problem described in Fig. 4.6, the number of variables is $\mathcal{O}(nm\rho d^2)$, as z is the variable with the most indices. Similarly, the number of constraints is also $O(nm\rho d^2)$ because of (C$1_5'''$). Note that ϕ does not influence the theoretical descriptive complexity of the MILP problem in Fig. 4.6, but it strongly limits the complexity reduction reachable by pre-processing methods.

Currently we have no NP-hardness proof of the MILP problem in Fig. 4.6. However, as solving MILP problems is in general an NP-hard problem [63], we expect the worst case complexity of our methodology to be exponential in any of the dimensions n, m, d, ρ and ϕ. If we fix all of the ϕ values in the parameter vector p, solving the resulting MILP problem is equivalent to running the corresponding LIBBR program for n iterations over each of the m training sets, which can be done in polynomial time $O(n\,m\,\rho\,d)$. This is a consequence of Th. 4.5.

By contrast, this no longer holds if any of the parameters are not fixed. Empirically, our methodology does not scale all that well as ϕ grows, as seen in Fig. 6.6 and Fig. 6.7. Luckily however, ϕ does not attain large values in practical cases from industry, as there cannot be too many changes to a process at a time. Usually, the value is $\phi \leq 3$.

While we use the standard CPLEX solver in our experiments in Chapter 6, a brute force approach would be to use the BR engine to test $O(2^{\phi})$ values of the vector p, and solve the BR for each of them. Noting $C \leq 2\,n,\ d\,(\rho + d)$ the complexity of an execution of the BR program as defined in Th. 4.5 (C is small by necessity in real-world applications), the complexity of this approach would be is thus $\mathcal{O}(C\,m\,2^{\phi})$. Since our MP based algorithm explores the configuration space implicitly, it is likely to be better than this brute force approach, which explores the configuration space explicitly.

Chapter 5

Theoretical framework for learning frequency distributions

A variation on the problem considered in previous sections is generalized statistical goal learning. While looking to approach a set average decision is useful, a lot of industrial problems have more specific requirements. Goals that may be considered as statistical are for example: 'Having at least 30% of loan requests accepted automatically, and at most 50% of loan requests rejected automatically'; or 'Having the statistical distribution of the age of accepted loan owners be within 5% of a normal distribution'. Formally, we call a learning problem with a function class $P(p, .), p \in \pi$ and a set of input data Q a statistical goal learning problem if the goal is to find a p such that $\tilde{Q} = P(p, Q)$ follows a statistical distribution $\mathscr{P}$ such that $\|g(\mathscr{P}(\tilde{Q})) - g_0\| < \epsilon$, where g is a real function of statistical distributions.

The most restrictive goal possible among such learning problems is when $g(\mathscr{P})$ is the statistical distance between $\mathscr{P}$ and a given distribution, in which case the problem is essentially to learn a function over probability distributions. On the other hand, having the goal be $g(\mathscr{P}(\tilde{Q})) = f(\mathbb{E}(\tilde{Q}))$ corresponds exactly to the learning problem explored in previous chapters.

Research related to learning probability distributions can be found in a different form, although learning a probability distribution without learning the specific probabilities of each data point has not been studied in ML before.

Many ML techniques naturally produce probabilistic classifiers when solving a classification problem [79]. Some of those classifiers are naturally biased, but can be corrected with proper calibration techniques [88]. Other ML algorithms such as Support Vector Machines produce classifier scores, which can also be transformed into probabilities using different techniques [81], Platt scaling being the best known one [97].

Another appearance of probabilities within the scope of ML is of course the application cases where the object to be learned is a probability distribution. Even then, the widely studied application case learns transition or relatedness probabilities in networks, such as social networks [60], which amounts to learning the individual probabilities of each transition.

For all statistical goal learning problems, the same general approach will also yield an acceptable learning algorithm: transforming the condition over the statistical distribution of the output into a set of MP constraints while using the constraints from Fig. 4.2 to model the execution of the BR program produces an optimization problem, which can for most forms of statistical goals be solved using standard methods in the case of LIBBR programs.

In this chapter, we examine a different formalization of the statistical goal learning problem which applies to all problems which model the goal using boxes to quantize the output of the BR program. In the first section, we formalize the general form of the problem and provide a MIP problem which is linear as long as the statistical constraints over the quantized output are linear. We then produce an example of an MP based learning algorithm for each of two problems that are representative of common requests among BR engine users, which both enter this category of problems: (1) the learning problem where the output value of a certain variable can match a given value in at most $g \times m$ of the cases, where $m = \mathrm{card}(Q)$; (2) the learning problem where the output values of a given variable must follow a distribution that is as close to uniform as possible, i.e. where $\forall s, t \in \{1, \ldots, N\}, \left|\nu_s - \nu_t\right| \leq 1$.

5.1 Learning quantized distributions

A common variation of the statistical goal learning problem is the quantized form. In this form, we divide the output space $P(\pi, X)$ into N intervals or boxes. We then use the number of outputs $P(p, q), q \in Q$ in these categories, noted as $\nu_1(p), \ldots, \nu_N(p)$, to define the statistical constraint, i.e. we have $g(\mathscr{P}(\tilde{Q})) = g'(\nu_1(p), \ldots, \nu_N(p))$. This corresponds broadly to cases where the relevant output has discrete values and to cases where the relevance of continuous outputs can be viewed in histograms. We can formalize this learning problem as:

$$\left.\begin{array}{r} \min\limits_{p} \|p - p^0\| \\ \mathscr{C}(\nu_1(p), \ldots, \nu_N(p)) \end{array}\right\} \tag{5.1}$$

with the same notations as in 2.2, and with $\mathscr{C}$ a constraint or set of constraints. While this formulation uses the number of outputs rather than the probabilities themselves, the relation between the two is simply a ratio of $1/m$, where $m = \texttt{card}(Q)$ is the number of training data points.

The MIP problem from Fig. 5.1 models the problem from Eq. 5.1 when the only relevant output is x_1. With the same notations as before, we also note $O = \{1, \ldots, N\}$, such that $\forall t \in O, \nu_t = \mathrm{card}\{j \in J \mid x^1_{n,j} \in [\beta_{t-1}, \beta_t]\}$. We enforce this definition of ν_t by using an auxiliary binary variable s_{tj} with the property: $s_{tj} = 1$ iff $x^1_{n,j} \in [\beta_{t-1}, \beta_t]]$. The other auxiliary binary variables s^U_{tj} and s^L_{tj} are used to enforce this property.

The constraints are mostly similar to the ones in Fig. 4.3. We simply add the goal of minimizing the variation of the parameter value and the constraints $\mathscr{C}(\nu_1(p), \ldots, \nu_N(p))$ from Eq. 5.1. The new constraints (C7$^{\text{hist}}$) through (C11$^{\text{hist}}$) represent the definition of $\nu_1, \ldots, \nu_N$.

That solving the MIP problem in Fig. 5.1 also solves the problem in Eq. 5.1 is a direct consequence of the fact that the constraints in Fig. 4.2 simulate $P(p, q)$. The proof is simple since (C7$^{\text{hist}}$) through (C11$^{\text{hist}}$) trivially represent the definition of $\nu_1, \ldots, \nu_N$. However, this formulation is still quite abstract, as it depends heavily on the form of $\mathscr{C}$. We do note that assuming $\mathscr{C}$ is linearizable, the MP problem is in fact linearizable using the transformations which have led from Fig. 4.3 to Fig. 4.6 in Chapter 4. The MP problem in Fig. 5.1 can almost always be simplified given a particular constraint over the quantized distribution, as we see in the rest of this part.

5.2 Upper bound on a specific output value's probability

One of the requests that are made of the business analysts maintaining a BR program is that the program respect an upper bound on the number of outputs which have a given value. In the loan approval example, we may have a BR program used to decide whether the bank will investigate the loan request further or simply accept the automated decision taken by an expert system. That BR program has a binary output value. This bank's high-level strategy requires that no more than 50% of loans are treated manually (because of the limited availability of bank managers, for example), but 60% of loans are currently flagged to be treated manually.

$$
\begin{array}{lll}
\underset{p,a,x,y,s,s^U,s^L,o}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\text{C1}), (\text{C2}), (\text{C3}), (\text{C4}), (\text{C5}), (\text{IC1}) & \\
 & \mathscr{C}(\nu_1, \ldots, \nu_N) & \\
\forall (t,j) \in O \times J & \beta_{t-1} - M(1 - s_{tj}) \leq x_1^{n,j} \leq \beta_t + M(1 - s_{tj}) & (\text{C7}^{\text{hist}}) \\
\forall (t,j) \in O \times J & x_1^{n,j} \geq \beta_t - M s_{tj}^U & (\text{C8}^{\text{hist}}) \\
\forall (t,j) \in O \times J & x_1^{n,j} \leq \beta_{t-1} + M s_{tj}^L & (\text{C9}^{\text{hist}}) \\
\forall (t,j) \in O \times J & s_{tj} \geq s_{tj}^U + s_{tj}^L & (\text{C10}^{\text{hist}}) \\
\forall t \in O & \nu_t = \sum_{j \in J} s_{tj} & (\text{C11}^{\text{hist}}) \\
\forall (i,j) \in I \times J & x^{i,j} \in X & \\
 & p \in \mathbb{R} & \\
\forall (i,j,r,k) \in I \times J \times R \times D & y_{ijr} \in \{0,1\} & \\
\forall (t,j) \in O \times J & s_{tj}, s_{tj}^U, s_{tj}^L \in \{0,1\} & \\
\forall t \in O & \nu_t \in \mathbb{N} &
\end{array}
$$

Figure 5.1: **Mixed-Integer Program solving Eq. 2.2**

This scenario can be formulated as:

$$
\left.\begin{array}{l}
\min\limits_{p} \|p - p^0\|_1 \\
\mathbb{E}_{q \in Q}\Big[P(p,q)\Big] \quad \leq \quad g
\end{array}\right\} \tag{5.2}
$$

where P has an output in $\{0,1\}$, using the same notations as before. We provide a MILP problem which is equivalent to solving this type of learning problem, when the BR program is linear and iteration bounded.

As in Section 4.4, we consider linear BR programs with a known bound $(n-1)$ on the number of iterations of the loop, with ρ rules and with a parameter $p = A_{1,1,1}$. The MILP problem described in Fig. 5.2 is a reformulation of Eq. 5.2. In the case of this problem, we can remove some superfluous variables, since only one of the ν_t is relevant. The resulting MILP problem is very similar to the one in Fig. 4.6, simply replacing the constraint involving ε by a simple inequality constraint. As before, an equivalent MILP problem can be found for p taking the place of a scalar in B_r, H_r or G_r.

Every constraint numbered as before fulfills the same role. The additional constraints are:

- $(\mathrm{C6}'_{\mathrm{hist}})$ represents the goal from Eq. 5.2, that is a constraint over the average of the final values of x. It replaces $\mathscr{C}(\nu_1, \ldots, \nu_N)$ and all the constraints used to define ν_t from the MIP problem in Fig. 5.1.

$$
\begin{array}{lcr}
\underset{p,a,x,y,y^U,y^L,w,z}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\mathrm{C1}''_1), (\mathrm{C1}''_2), (\mathrm{C1}'''_3),(\mathrm{C1}'''_4), (\mathrm{C1}'''_5), (\mathrm{C2}), (\mathrm{C3}'') & \\
 & (\mathrm{C4}'_1), (\mathrm{C4}'_2), (\mathrm{C4}'_3),(\mathrm{C5}), (\mathrm{IC1}), (\mathrm{IC2}), (\mathrm{IC3}), (\mathrm{IC4}) & \\
 & \sum_{j \in J} x_1^{n,j} \leq mg & (\mathrm{C6}'_{\mathrm{hist}}) \\
\forall (i,j,r,h,k) \in I \times J \times R \times D^2 & x^{i,j}, z_{ijrhk} \in X & \\
\forall (i,j,r) \in I \times J \times R & w_{ijr} \in \mathbb{R}^d & \\
\forall r \in R & a_r \in \{0,1\}^{d \times d} & \\
 & p \in \{0,1\} & \\
\forall (i,j,r,k) \in I \times J \times R \times D & y_{ijr}, y^U_{ijrk}, y^L_{ijrk} \in \{0,1\} &
\end{array}
$$

Figure 5.2: **MILP Formulation for Solving Eq. 5.2**

Solving the MILP problem in Fig. 5.2 solves the learning problem in Eq. 5.2 when the BR program is LIBBR. The proof directly derives from Th. 4.10, as it is the exact same MILP problem using $(\mathrm{C6}'_{\mathrm{hist}})$ instead of $(\mathrm{C6}'')$. The problem it solves is almost the same as the problem solved by the MILP problem in Fig. 4.6, i.e. Eq. 2.2 with the difference being that the constraint is not $\left|\mathbb{E}_{q \in Q}\left[f\Big(P(p,q)\Big)\right] - g\right| \leq \varepsilon$ but $(\mathrm{C6}'_{\mathrm{hist}})$: that problem is exactly Eq. 5.2.

5.3 Almost uniform distribution

Another common statistical goal learning problem found among BR users consists of needing the output values to be distributed in a fashion close to the uniform distribution, for integer outputs. In the loan example, a bank might use a BR program to accept, reject, or assign a bank manager to the loan request, and therefore has a trinary return value, represented by an integer in $\{0, 1, 2\}$. That bank's strategy requires that the proportion of each output is $\{1/3, 1/3, 1/3\}$, but it is currently $\{1/4, 1/4, 1/2\}$.

This scenario can be formalized as:

$$\left.\begin{array}{r}\min\limits_{p,x} \|p - p^0\|_1 \\ \forall t, \tau \in \{1, \ldots, N\}, \quad \left|\nu_t - \nu_\tau\right| \quad \leq 1\end{array}\right\} \tag{5.3}$$

Note that the solution to this problem is not always a truly uniform distribution, simply because there is no guarantee that m is divisible by N. However, it will always be as close as possible to a uniform distribution, since the constraint imposes that all the outputs will be reached by either `floor`(m/N) or `ceil`(m/N) data points. Again, we use numbers of outputs instead of true probabilities without loss of information, which allows us to use the fact that the former are integers. We provide a MILP problem which is equivalent to solving this type of learning problem, when the BR program is linear and iteration bounded.

We again consider linear BR programs with a known bound $(n - 1)$ on the number of iterations of the loop, with ρ rules and with a parameter $p = A_{1,1,1}$. The MILP problem in Fig. 5.3 solves Eq. 5.3. As before, an equivalent MILP problem can be found for p taking the place of a scalar in B_r, H_r or G_r.

Every constraint numbered as before fulfills the same role. The additional constraints are:

- ($\mathrm{C7}''_{\mathrm{hist}}$) through ($\mathrm{C9}''_{\mathrm{hist}}$) are the adaptation of ($\mathrm{C7}_{\mathrm{hist}}$) through ($\mathrm{C11}_{\mathrm{hist}}$) to the relevant case of integer outputs

- ($\mathrm{C6}''_{\mathrm{hist}}$) represents the equivalent to $\mathscr{C}$ from Eq. 5.1.

This MILP problem is obviously equivalent to solving Eq. 5.3, since it is for the most part a straight linearization of the MIP problem in Fig. 5.1. A more intuitive proof consists of again comparing this MILP problem to the one in Fig. 4.6, and remarking that the MILP problem in Fig. 5.3 solves:

$$\left.\begin{array}{r}\min\limits_{p,x} \|p - p^0\|_1 \\ (\mathrm{C6}''_{\mathrm{hist}}), (\mathrm{C7}''_{\mathrm{hist}}), (\mathrm{C8}''_{\mathrm{hist}}), (\mathrm{C9}''_{\mathrm{hist}})\end{array}\right\}$$

which is equivalent to Eq. 5.3.

$$\boxed{\begin{array}{lcr}
\underset{p,b,x,y,y^U,y^L,w,s,s^L,s^g}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\mathrm{C1}''_1), (\mathrm{C1}''_2), (\mathrm{C1}'''_3), (\mathrm{C1}'''_4), (\mathrm{C1}'''_5), (\mathrm{C2}), (\mathrm{C3}') & \\
 & (\mathrm{C4}'_1), (\mathrm{C4}'_2), (\mathrm{C4}'_3), (\mathrm{C5}), (\mathrm{IC1}), (\mathrm{IC2}), (\mathrm{IC3}), (\mathrm{IC4}) & \\
\forall (t,\tau) \in O^2 & -1 \le \nu_t - \nu_\tau \le 1 & (\mathrm{C6}''_{\mathrm{hist}}) \\
\forall (t,j) \in O \times J & t - M(1 - s_{tj}) \le x_1^{n,j} \le t + M(1 - s_{tj}) & (\mathrm{C7}''_{\mathrm{hist}}) \\
\forall (t,j) \in O \times J & x_1^{n,j} \ge t - M s_{tj}^U & (\mathrm{C8}''_{\mathrm{hist}}) \\
\forall (t,j) \in O \times J & x_1^{n,j} \le t + M s_{tj}^L & (\mathrm{C9}''_{\mathrm{hist}}) \\
\forall (i,j) \in I \times J & x^{i,j} \in X & \\
\forall r \in R & a_r \in \{0,1\}^{d \times d} & \\
 & p \in \{0,1\} & \\
\forall (i,j,r,k) \in I \times J \times R \times D & y_{ijr}, y_{ijrk}^U, y_{ijrk}^L \in \{0,1\} & \\
\forall (t,j) \in O \times J & s_{tj}, s_{tj}^U, s_{tj}^L \in \{0,1\} & \\
\forall t \in O & \nu_t \in \mathbb{N} &
\end{array}}$$

Figure 5.3: **MILP Formulation for Solving Eq. 5.3**

Chapter 6

Experimental work

The existence of an algorithm which can solve the most common of statistical goal learning problems on LIBBR programs is the most important result of this thesis. From the point of view of BRMS vendors such as IBM, however, the possibility of applying this algorithm to their clients' BR programs as an integral and automatic part of their product is also a crucial part of what this thesis was meant to do.

In this chapter, we first present the Proof of Concept (POC) Java program which was developed during this thesis. It is a simple parser which reads a set of BRs in ODM Business Rules archive format and provides the MILP problem in AMPL format in a couple of files. A simple AMPL script can then solve the MILP problem using existing solvers, such as CPLEX for example. We then use the experimental data obtained by solving the MILP problem our algorithm associates with randomly generated LIBBR programs (see Fig. 4.6) to gain some insight into the properties of this algorithm in terms of accuracy, scalability and complexity. Finally, we briefly look at the performance of the MILP problems associated with the maximum percentage problem in Sect. 5.2 (Fig. 5.2) and the almost uniform distribution problem in Sect. 5.3 (Fig. 5.3).

6.1 Parsing ODM

One of the most interesting points about the algorithm presented in Chapter 4.4 is that it can be automated. The pipeline in Fig. 6.1 shows the organization such an automated learning process would have. The input of the statistical goal learning problem (BR program, statistical goal and learning data) is first transformed into an optimization problem, which is then solved using a standard solver. Each step in this pipeline can be automated. We provide a Proof of Concept (POC) program demonstrating this automation for the first step of transforming the BR program into constraints in the case of a LIBBR program written in ODM.

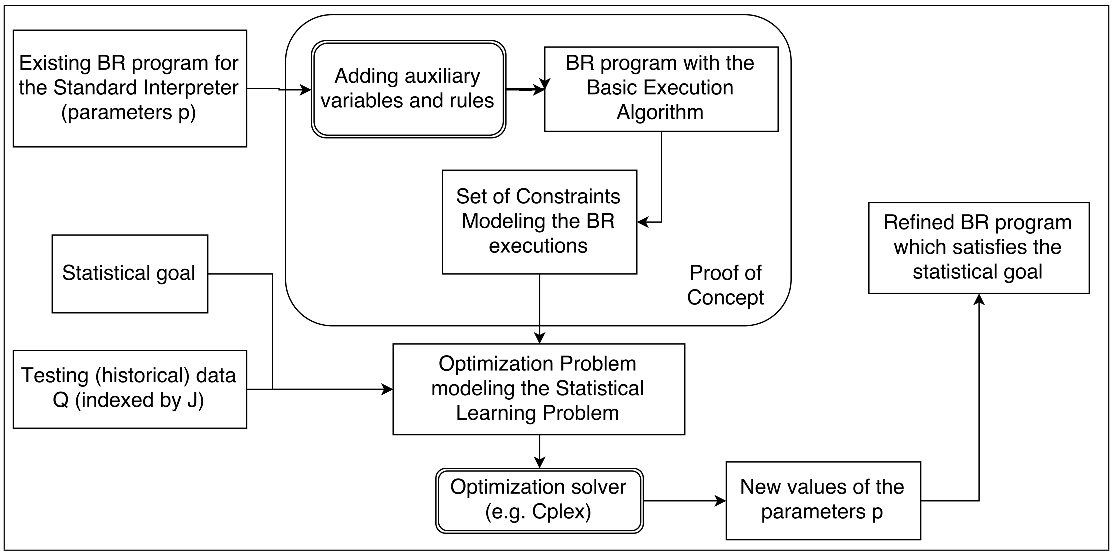

Existing BR program for the Standard Interpreter (parameters p)
Adding auxiliary variables and rules
BR program with the Basic Execution Algorithm
Set of Constraints Modeling the BR executions
Proof of Concept
Statistical goal
Refined BR program which satisfies the statistical goal
Testing (historical) data Q (indexed by J)
Optimization Problem modeling the Statistical Learning Problem
Optimization solver (e.g. Cplex)
New values of the parameters p

Figure 6.1: Execution pipeline for an entirely automated application of the statistical goal learning algorithm

As the MILP problem to solve has a fixed form, it can be written as a fixed *mod* file written using the standard AMPL syntax [6]. The Proof of Concept (POC) Java program we created takes as input an ODM Business Rules archive in *jar* form, and gives as output the *dat* file containing the correctly formatted input to this *mod* file, with the input data Q being randomly generated – making the program parse another file containing preset values of Q has been partially programmed, but the lack of industrial data to parse has made the point moot.

We suppose for simplicity of the POC that the ODM BR Project follows the following conditions:

- it uses only primitive Java types in its ODM parameters
- the input variables are defined as IN/OUT or OUT ODM parameters; the parameters are coded as IN ODM parameters (ODM parameter is the name used in ODM for any value that is not fixed before compile-time, when writing the ruleset)
- each rule is linear
- each rule's condition has only conjunctions; any disjunction must be written as two rules
- each rule's base conjunct that uses a Binary operator such as $=$ or $\leq$ must have the variable on the left-hand side and the values or parameters on the right-hand side

The resulting RulesetArchive file is obtained in ODM 8.5.1 by right-clicking the BR project and choosing the "Export$\mapsto$Ruleset Archive" option, from the Rules perspective.

Note that the resulting output file cannot directly be used by AMPL, since it is missing both the testing set inputs Q and the goal g. However, these values can simply be directly appended to the resulting file in text format using the AMPL grammar. Additionally, we are unable to parse the default values of the ODM parameters, thus the p^0 value of any parameter must be added by modifying the *dat* file. In the absence of such a modification, those values are assumed to be randomly generated.

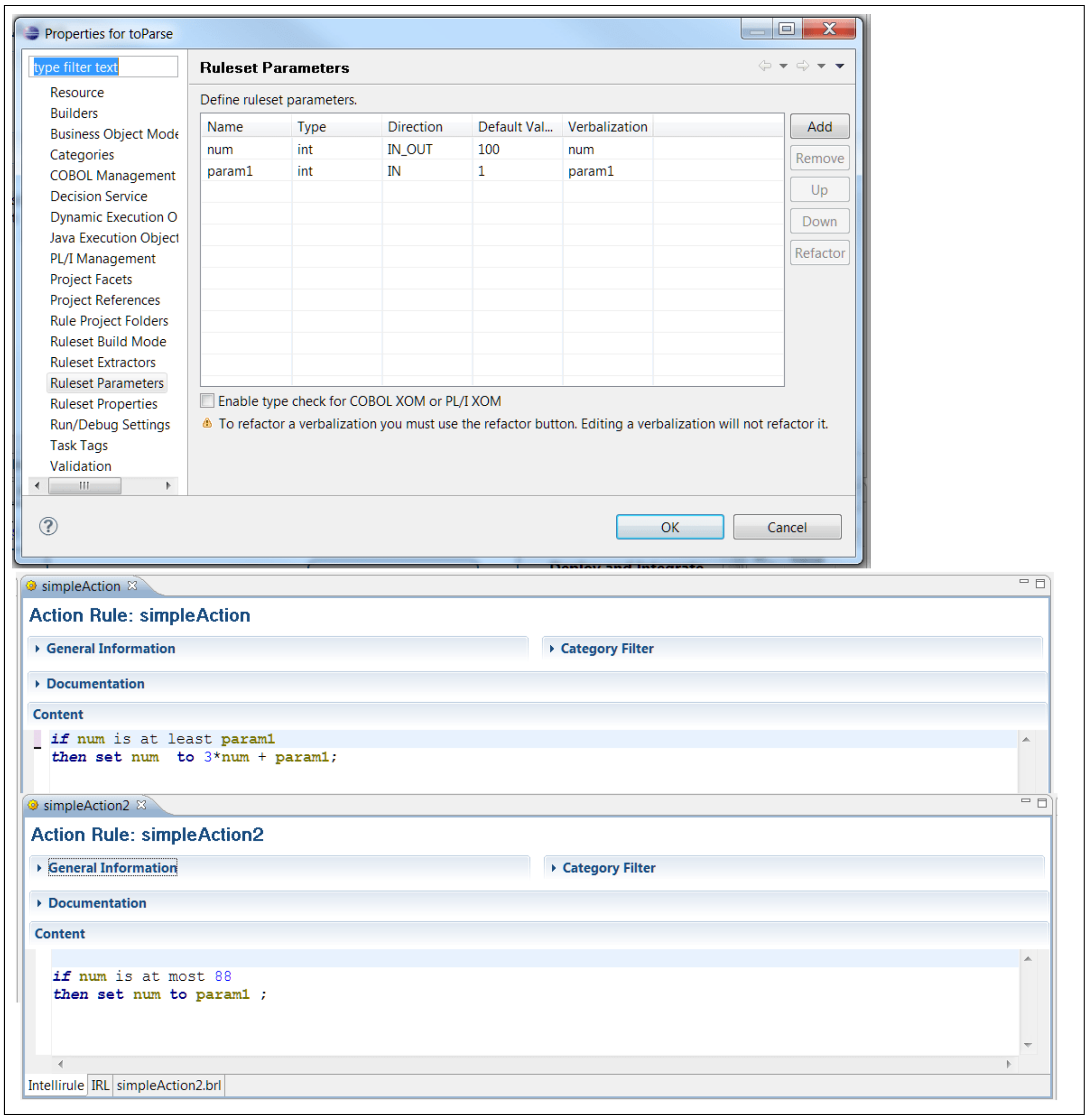
The example program displayed in Fig. 6.2 is very simple, as it includes only one integer variable and one integer parameter. The resulting *dat* file is displayed in Fig. 6.3. Note that the OPL file has twice as many rules because of the Refraction clause in $\mathscr{I}_S$: we double each rule in the BR program to obtain a program which is meant to be executed in $\mathscr{I}_0$, as explained in the proof of Th. 4.1. Similarly, there is one additional variable per rule because of this same clause. Note that we assume that the implicit priority is absolute in our parsing, making the Recency clause inactive.

Our method is efficient as it simply uses the existing infrastructure of ODM IlrRulesetArchive and IlrWriter classes to explore the BR program (for our chosen example, less than a second). The variables and parameters are created in advance by using the structure embedded in the ODM archive, then the POC explores each condition and action to obtain the values of the relevant OPL parameters, while keeping the integer and non-integer variables separate:

- `LB` in the *dat* corresponds to L in Fig. 4.5
- `HB` in the *dat* corresponds to H in Fig. 4.5
- `A` in the *dat* corresponds to A in Fig. 4.5
- `B` in the *dat* corresponds to B in Fig. 4.5

An important part of the POC was automating the transition from integer variables to binary variables, which are necessary for the MP problem to be linearizable. This is done by transforming A into a tensor rather than a matrix, making `A[r,i,j,k]` a vector corresponding to the value of $A_{r,i,j,k}$ in binary representation.

This POC can be developed further by integrating more of the native functionalities of ODM which make it so interesting, regardless of the limitations of the learning algorithm. In particular, the use of non-primitive variables can be done



Ruleset Parameters

Define ruleset parameters.

Name	Type	Direction	Default Val...	Verbalization
num	int	IN_OUT	100	num
param1	int	IN	1	param1

☐ Enable type check for COBOL XOM or PL/I XOM

⚠ To refactor a verbalization you must use the refactor button. Editing a verbalization will not refactor it.

Action Rule: simpleAction

```
if num is at least param1
then set num  to 3*num + param1;
```

Action Rule: simpleAction2

```
if num is at most 88
then set num to param1 ;
```

Figure 6.2: Screenshots of the ODM 8.5.1 Eclipse environment displaying a simple BR program

by making use of more advanced ODM developer tools. While this is of extreme interest for industrial applications, it is not the purview of this thesis.

Another avenue of development is by adding more complex elements such as Recency or recognition of default parameter values in the RulesetArchive. These are developer tasks. Additionally, a number of more industrially relevant advances might make it into such a POC. In particular, automated transformation of disjunctions in BR conditions into multiple BRs (see Fig. 6.4), automated use of additional boolean variables to compute conditions such as:

```
**if** $\text{num} \geq 3 \land \text{num} \geq \text{param1}$ **then**
**end if**
```

or direct inclusion of the goal and testing input set are all feasible features to add.

6.2 Experimental data on learning LIBBR

In this section, we use the experimental data obtained by solving the MILP problem our algorithm associates with randomly generated LIBBR programs (see Fig. 4.6) to gain some insight into the properties of this algorithm in terms of accuracy, scalability and complexity. We also look at the performance of the MILP problems associated with the maximum percentage problem in Sect. 5.2 (Fig. 5.2) and the almost uniform distribution problem in Sect. 5.3 (Fig. 5.3).

In the first subsection, we describe in detail the setup of the experiments, notably the generation of the BR programs. In the second subsection, we explore the accuracy of the algorithm by trying to vary the value of the ε which appears in Eq. 2.2. We then check on various elements linked to scalability, from the number of rules ρ to the bound on the number of iterations of the loop n. We complete the chapter with some data taken from running the MILP problems associated with other statistical goal learning problems.

6.2.1 Experimental setup

Randomly generated BR programs have been obtained from a Python script. We define the space X as $X \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{Z}$. The BR programs are sets of a variable number ρ of linear BRs:

```
**if** $L_r \leq x \leq H_r$ **then**
    $x \leftarrow A_r x + B_r$
**end if**
```

where L_r, H_r, B_r are vectors of scalars in $[-5, 5]$; $L_r \leq H_r$ and A_r are $d \times d$ matrices of binary variables. We call d the dimension of the BR program. We also use n the upper bound on the number of iterations of the execution loop (see Sec. 4.1) and m the size of the training set (see Sec. 2.2). The instances are vectors q_j with values in $[-5, 5]$. All values are generated using a uniform distribution. We use a variable value of ε, as it appears in Eq. 2.2. For each BR program, we try to obtain a goal $g = 0$ by optimizing over ϕ randomly chosen parameters.

We randomly generate samples of 100 linear BR programs and corresponding sets of instances with $d = 3$, $n = 10$ and $m = 100$.

We use these BR programs to study the computational properties of the MILP problem. The value of M used is customized according to each constraint, and is ultimately bounded by 51 (strictly greater than five times the range of possible values for x). We write the MILP problem as an AMPL model, and solve it using the CPLEX solver version 12.6.1.0 on a Dell PowerEdge 860 running CentOS Linux, with 4 CPU cores of frequency 2400MHz with a RAM of 8GB.

6.2.2 Validating the learning algorithm

We test the validity of the proposed learning algorithm by solving the MILP problem obtained from many randomly generated BR programs with varying values of ε.

We fix the number of rules to $\rho = 4$ and so the number of parameters is 20. This is much lower than the number of rules used by BR programs destined to industrial usage. We observe the average solving time and the proportion of solvable MILP problem instances among all instances for different values of ε. For ease of comparison, we do not generate a different set of BR programs for each value of ε. Instead, the tests in this subsection are executed on a single set of one hundred BR programs for different statistical goals: the MILP problems we solve have the same values of $L_r \leq H_r$, A_r and B_r but different values of ε.

For each BR program, we start with $\varepsilon = 1$. This corresponds to a very high tolerance (20 % of the range of possible values). If the instance is solvable, we decrease ε using $\varepsilon \leftarrow \varepsilon - 0.2$ until we reach an unsolvable instance; otherwise we increase ε using $\varepsilon \leftarrow 1.5\varepsilon$ until we reach a solvable instance or $\varepsilon \geq 5$, whichever happens first.

An instance is considered unsolvable if it is infeasible or it has no integer solution after fifteen minutes (900s) of solver time.

Fig. 6.5 shows the proportion of solvable instances as a function of ε. Being careful of the nonlinear scale of the figure, ε seems to have a greater influence on solvability when small and a reduced influence as it grows. In other words, our method will relatively easily find the best parameter in most cases, but difficult problems remain difficult even when allowing for a greater distance to the desired average f.

Furthermore, with random BR problems that include both unsolvable and already optimal situations, we solve 50 percent of problems with $\varepsilon = 0.4$ which means allowing $\mathbb{E}(f) \in [-0.4, 0.4]$. This is too low for industrial applications, but approaching the desired average by 8 percent of the possible range in half the cases is a promising start. Our method as it is described cannot currently be used as a general tool as too many BR programs cannot be solved accurately.

The restriction to $n = 10$ for those orders of ρ accurately models real business processes where BRs rarely loop. The sample size of $m = 100$ is much lower than would be realistic, as the training data must be statistically representative of the expected input value distribution. The dimension $d = 3$ is arbitrary and much lower than can be expected in actual business processes, where dimensionality can be over an order of magnitude higher.

6.2.3 Testing performance

Using the same source of randomly generated BR programs, we try to look at the performance of the MILP problem in Fig. 4.6. We set a fixed ε value of 1, which is the highest tolerance used in the previous subsection.

A naive approach is to believe that BR programs can be learned entirely. In this approach, the number of parameters ϕ increases proportionally with the number of rules ρ, as each rule can be parametrized separately. We suppose that the number of parameters is $\phi = 5\rho$.

We observe the average solving time and optimal objective for different values of ρ (Tab. 6.1) among the solvable MILP instances. An instance is considered unsolvable if it is infeasible or it has no integer solution after one hour (3600s) of solver time.

Value of ρ	Proportion of instances solvable in an hour	Average solver times over solvable instances	Average objective values over solvable instances
1	1.00	2.09	0.98
2	0.98	22.98	2.14
3	0.96	265.35	4.04
4	0.89	737.67	6.98
5	0.66	929.32	7.77

Table 6.1: Experimental values for the scalability of the MILP method

While it could be argued that the increase in the number of parameters has an obvious effect over the difficulty of the problem, the study of an increase in ρ without the proportional increase in ϕ leads to a drastic and predictable increase in infeasible instances. Even with our setup, the proportion of solvable instances is lower as ρ increases, although that is mostly due to the solver exceeding the time limit. As a high value of ρ is the main issue when scaling up to industrial BR programs, it still seems worth studying.

Unfortunately, we observe that the direct solving of the MILP problem described in Section 4.4.2 is not practical for learning all or even a majority of the parameters in industrial-sized BR programs. Furthermore, the increase in computational time is not linear with the number of ρ, but rather exponential as seen on Fig. 6.6. The increase in the optimal objective value is intuitive and does not seem drastic, which indicates that the experimental setup is somewhat realistic. Luckily, industrial BRs do not need to be parametrized entirely – oftentimes, the number of control parameters is simply 1!

We now wish to consider the influence of each characteristic of a statistical goal learning problem applied to an iteration bounded linear BR program on the computational complexity of the MILP problem generated by our learning algorithm, irrespective of its feasibility or accuracy.

We again randomly generate 100 LIBBR programs for each value of the problem characteristics, and display the average solving time taken by CPLEX to solve the resulting MILP problems. Note that this does not take into account the time taken by the automated pre-processing done by CPLEX (‘presolve’). While the pre-processing can at times be long, it does not accurately reflect the complexity of the learning since the model given to CPLEX is written in a very general way, with all meaningful information conveyed by the data file. This makes the pre-processing more costly than it would be in practical applications, since tasks such as defining the values in L, H, A, b that are not parameters as constants would otherwise be done beforehand.

We have chosen the values of the problem characteristics to be somewhat realistic, with at least some feasible MILP problems in each case. For each BR program, we have the common characteristics of chosing values in $[-5, 5]$, with $\varepsilon = 1$, and the number of integer dimensions of X being $d_{int} = 1$.

Fig. 6.7 shows the evolution of the solving time with ϕ. We have fixed the values of the other problem characteristics at $n = 10$, $m = 100$, $d = 3$, $\rho = 4$, and make ϕ vary from 1 to 12. We observe an exponential growth in line with our theoretical expectations from Sec. 4.5. In particular, we observe that for $\phi \leq 5$, the average solving time is less than a second, which is very much realistic for real-world applications.

Fig. 6.8 shows the evolution of the solving time with n. We have fixed the values of the other problem characteristics at $m = 100$, $d = 3$, $\rho = 4$, $\phi = 1$, and make n vary from 6 to 14. We observe an exponential growth, which is a concern but expected (at least in the worst case) when using the default solver settings, as explained in Sec. 4.5. However, given the fact that most industrial BR programs satisfy $n \leq \rho$ (because of decision tables), we can see that n should not be the cause of unpractical average solving times in real-world applications.

Fig. 6.9 and 6.10 show the evolution of the solving time with m. We have fixed the values of the other problem characteristics at $m = 100$, $d = 3$, $\rho = 4$, $\phi = 1$, and make n vary from 6 to 14. We observe an irregular growth which is in part explained by the lack of feasible MILP problem for values $m \geq 500$. For both $p = 1$ and $p = 3$, a single MILP problem is feasible at $m = 2000$, which explains the drop in average solving time. Notably, none of the MILP problems we tried to solve reached the upper limit on solving time (3600s, i.e. one hour). The stagnation in solving time for $p = 1$ is probably due to the lack of feasible BRs (only seven BRs over the whole curve are feasible). We do observe an exponential growth for as long as there are feasible BRs on the other curves, more obviously on Fig. 6.10.

Fig. 6.11 shows the evolution of the solving time with d, to be precise we vary

d_{float} the number of real-valued dimensions of X. We have fixed the values of the other problem characteristics at $n = 10$, $m = 100$, $\rho = 4$, $\phi = 1$, and make d vary from 3 to 5. We observe a stable, slightly decreasing curve which is very promising as d is one of the most variable characteristics of industrial BRs, depending strongly on the application. It correlates with an increase in feasible instances and a sharp decrease in nontrivial instances, indicating that this decreasing curve is due to a greater number of the randomly generated BRs being already optimized.

Fig. 6.12 shows the evolution of the solving time with ρ. We have fixed the values of the other problem characteristics at $n = 10$, $m = 100$, $d = 3$, $\phi = 5$, and make n vary from 4 to 10. We observe an exponential growth which stabilizes for values of $\rho \geq 12$, despite the fact that the values for which no MILP problem has a feasible solution are $\rho \geq 9$. This indicates a sort of higher bound above which ρ does not influence the computational complexity anymore, maybe because the increased number of rules is matched by an increased number of rules eliminated by pre-processing. However, as this threshold is greater than the one limiting the existence of feasible solutions, this has no direct interest for industrial applications.

We additionally try to correlate the average solving time with the ratio n/ρ, i.e. the ratio of the number of rules executed (or an upper bound thereof, at least) over the number of rules written in the BR program. For each of four values of this ratio, we run the tests for four values of ρ. A simple approximation of this ratio for industrial applications is to take the average width of the decision tables found in the BR program, since a decision table is simply a series of mutually exclusive rules, such as in Fig. 6.13. We display the results in Fig. 6.14, with the time displayed being the average solving time divided by 2^n. This division contracts the points obtained when running the tests, so that for each value of the ratio the solving time over 2^n is constant. This in turn means that n dominates the solving time, to the point where the value of ρ could be considered to only matter insofar as it helps determine n.

6.2.4 Testing other statistical goal learning problems

In Chapter. 5, we have proposed MILP problems for solving two examples of learning problems of a different form. We examine their behavior in some test cases to have an idea of whether the MP approach is appropriate for those statistical problems.

We randomly generate samples of 100 instances of linear BR programs for each of the problems from Eq. 5.2 and Eq. 5.3. BR programs corresponding to the problem in Eq. 5.2, resp. Eq. 5.3, are called P_1, resp. P_2. Each sample corresponds to a different number of control parameters c, each instance having a corresponding set of randomly generated inputs with $d = 3$, $n = 10$ and $m = 100$. The number of control parameters serves as an approximation of the complexity of the BR program to optimize: a more complex program will have more buttons to adjust, thus increasing the complexity, yet be more likely to have the goal be reachable at all, i.e. have the MILP problem be feasible. We define the space X as $X \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{Z}$. The BR programs are sets of $\rho = 10$ rules, where L_r, H_r, B_r are vectors of scalars in an interval `range` and A_r are $d \times d$ matrices of binary variables. In P_1, we use `range` $= [0, 1]$ and in P_2, we use `range` $= [0, 3]$. All input values q are generated using a uniform distribution in `range`.

We use these BR programs to study the computational properties of the MILP problems in Fig. 5.2 and Fig. 5.3. The value of M used is customized according to each constraint, and is ultimately bounded by 6 and 16 in P_1 and P_2 respectively (strictly greater than five times the range of possible values for x). We write the MILP problem as an AMPL model, and solve it using the CPLEX solver on a Dell PowerEdge 860 running CentOS Linux.

We observe the proportion of solvable instances of P_1 for c between 5 and 10 and $c = 15$ in Tab. 6.2. We use the MILP problem in Fig. 5.2 to solve Eq. 5.2 with the goal set to $g = 0.5$.

An instance is considered solvable if CPLEX reports an integer optimal solution or a (non-)integer optimal solution. We separate the instances where the optimal value is 0 from the others, as those indicate that the randomly generated BR program already fulfills the goal condition. We expect around fifty of those for any value of c.

In Fig. 6.15 and Fig. 6.16, we observe the success rate and the average solving time when considering only the non-trivial, non-timed out instances of P_1. The success rate increases steadily, as expected. The solving time seems to indicate a non-linear increase for c greater than 6, even with its values being somewhat unreliable due to the small sample. Knowing that average industrial BRs are more complex than our toy examples, regularly having thousands of rules, this approach to the Maximum Percentage problem does not seem applicable to industrial cases.

Number of control parameters c	5	6	7	8	9	10	15
Trivial solvable instances (objective $= 0$)	52	53	49	49	58	48	46
Non-trivial solvable instances (objective $\neq 0$)	5	6	5	13	6	6	8
Infeasible instances	43	43	40	36	31	35	14
Timed out instances	0	0	7	2	5	11	32

Table 6.2: **Experimental Values for the Max Percentage problem**

We now study the experimental results collected from testing the MILP problem in Fig. 5.3, which corresponds to the problem in Sec. 5.3. We observe the proportion of solvable instances of P_2 for c between 5 and 10 and $c = 15$ in Tab. 6.3. We use the MILP problem in Fig. 5.3 to solve Eq. 5.3 with $N = 2$. Again, we separate instances where the goal is already achieved before optimization, identifiable by being solved quickly with a value of $p = p^0$, i.e. an optimal value of zero.

In Fig. 6.17 and Fig. 6.18, we display the success rate and average solving time over the non-timed out, non-presolved instances for all three values of c. We observe a sharply non-linear progression, with the average problem taking about nine minutes with 15 control parameters. Knowing that average industrial BRs are much more complex than our toy examples, regularly having thousands of rules, we conclude that this method can only be used infrequently, if at all.

Number of control parameters c	5	6	7	8	9	10	15
Trivial solvable instances (objective = 0)	8	2	1	1	4	7	4
Non-trivial solvable instances (objective $\neq 0$)	9	2	8	5	4	15	32
Infeasible instances	83	96	91	93	92	77	63
Timed out instances	0	0	0	1	0	1	1

Table 6.3: **Experimental Values for the Almost Uniform Distribution problem**

```
% These are fixed by the ruleset
param num_rules      := 4;   param num_dim_int := 3;
param num_dim_float := 0;    param num_bits_A := 2;

param LB_int : 1 2 3 :=      param HB_int : 1 2 3 :=
  1    92  0 0                 1   101 0 1
  2    -1  1 0                 2   101 0 1
  3    -1  0 0                 3   88  1 0
  4    -1  0 1;                4   101 1 1;
param A_int :=
 [1,1,*,*]: 1 2 3 :=         param B_int :         % ETC.
  1    1    0 0               param LBparam_int : % ETC.
  2    0    0 0               param HBparam_int : % ETC.
  3    0    0 1               param Aparam_int := % ETC.
  % ETC.                      param Bparam_int :  % ETC.
 [4,2,*,*]: 1 2 3 :=
  1    0    0 0
  2    0    0 0
  3    0    0 0;

param LB_float;               param HB_float;
param A_float;                param B_float;
param LBparam_float;          param HBparam_float;
param Aparam_float;           param Bparam_float;
param M1 := 101.0;            param M2 := 201.0;
param M3 := 301.0;            param M4 := 401.0;
param M5 := 501.0;

% These are overwritten by the testing set input data
param max_iter := 5;          param num_exec := 6;
param goal      := 3.0;       param goal_type := 1;
param epsilon   := 10.0;
param input_int : 1 2 3 :=
  1    56  0 0
         % ETC.
  6    67  0 0;
param input_float;
```

Figure 6.3: Abridged display of the output of the ODM-to-OPL parser run over the ODM ruleset in Fig. 6.2

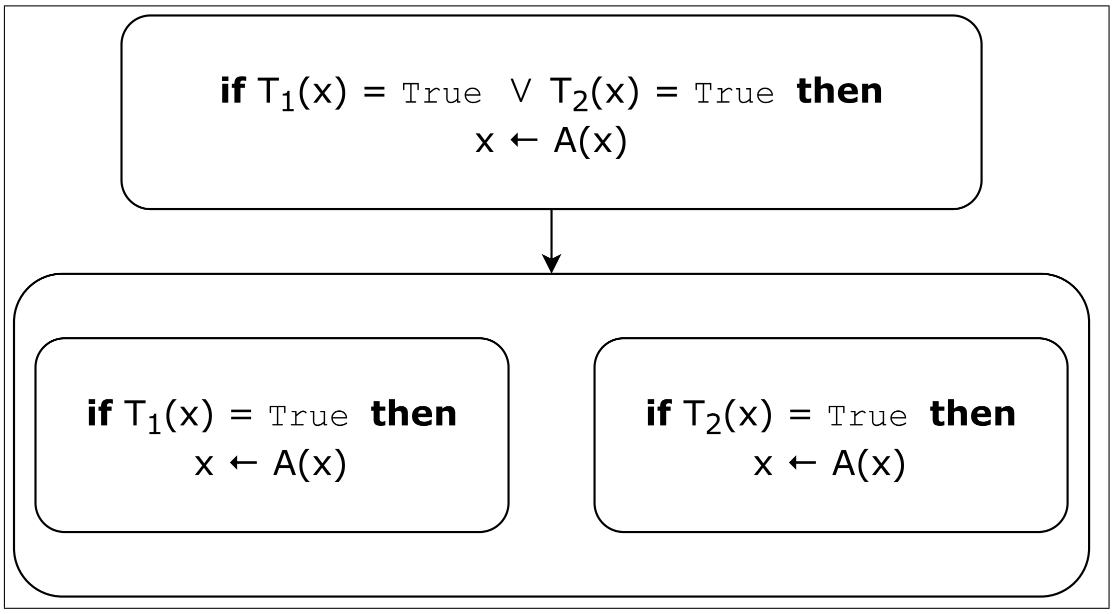


Figure 6.4: Transforming a BR with disjunctions into multiple conjunction-only BRs

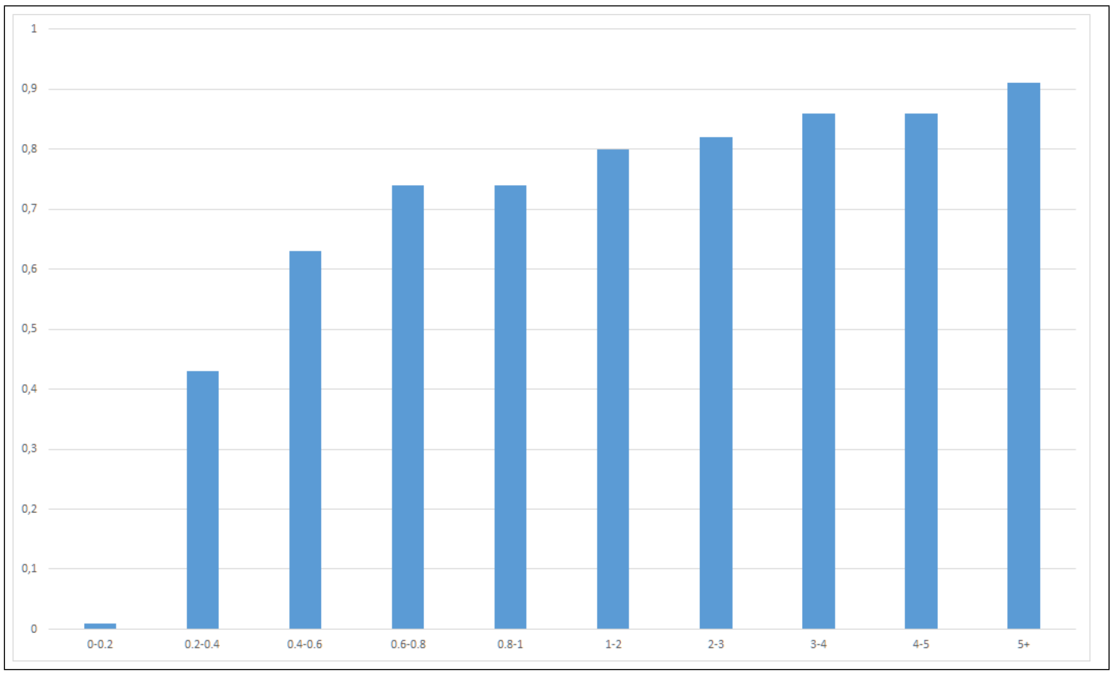


Figure 6.5: Proportion of solvable instances for varying values of ε

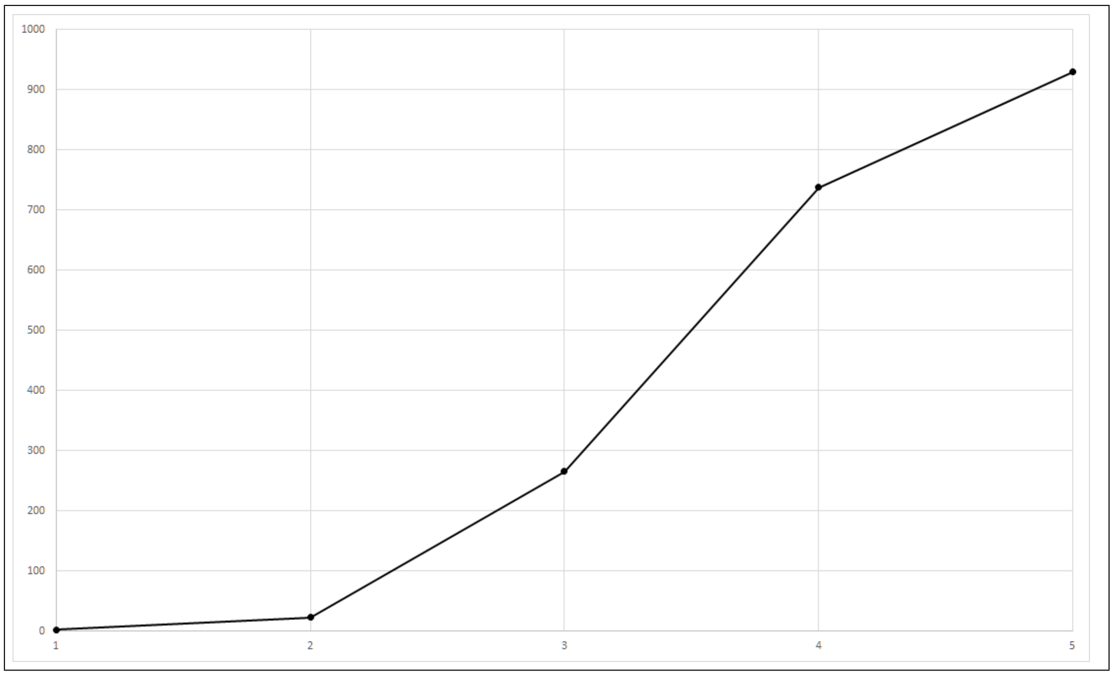


Figure 6.6: Variation of computational time (in seconds) with ρ (the number of rules in the BR program)

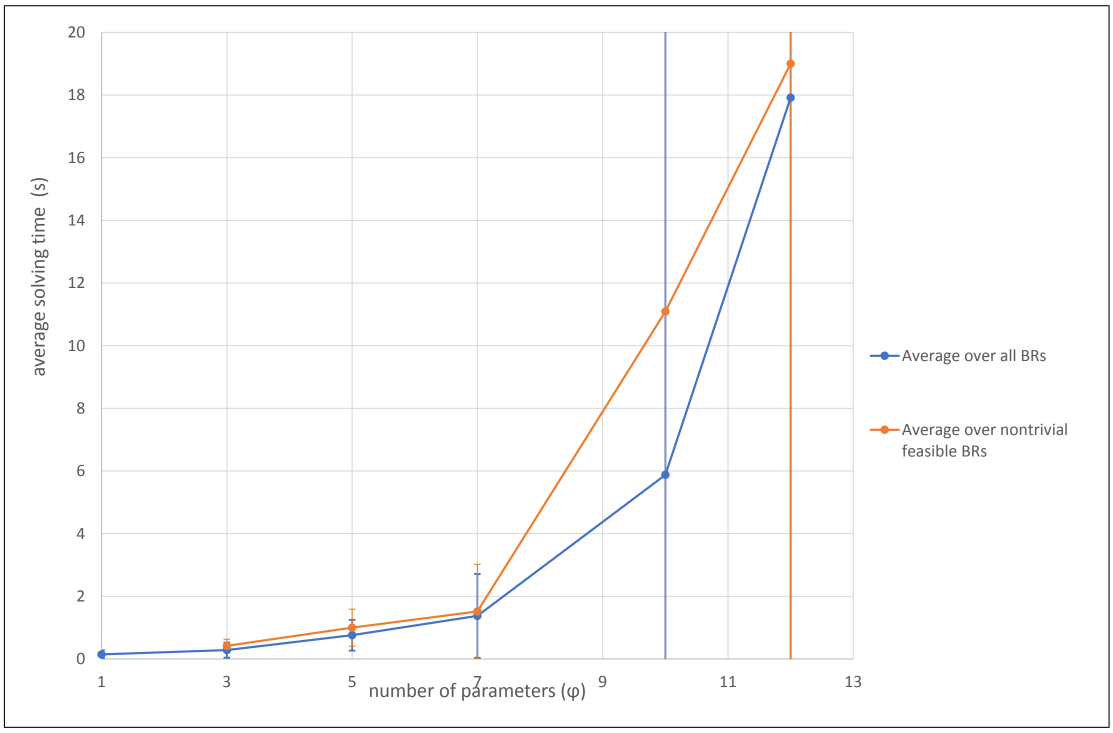


Figure 6.7: **Average solving time** over a range of values of ϕ (number of parameters to learn) for $n = 10$, $m = 100$, $d = 3$, and $\rho = 4$, with error bars representing the standard deviation

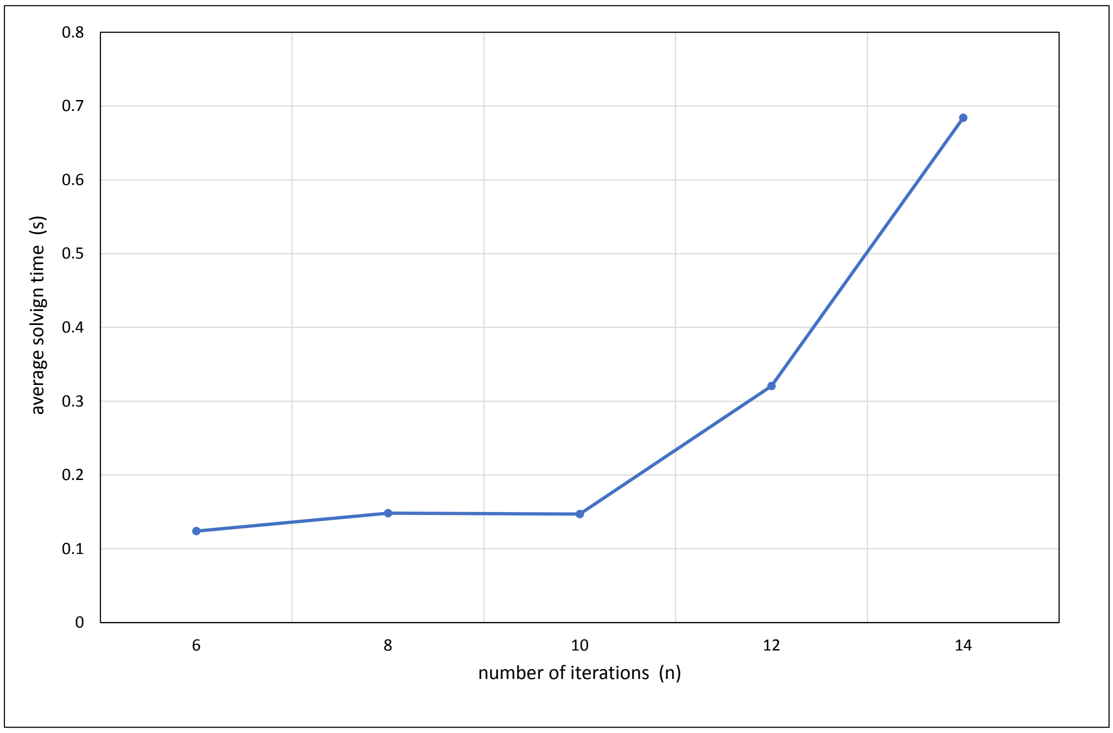


Figure 6.8: **Average solving time** over a range of values of n

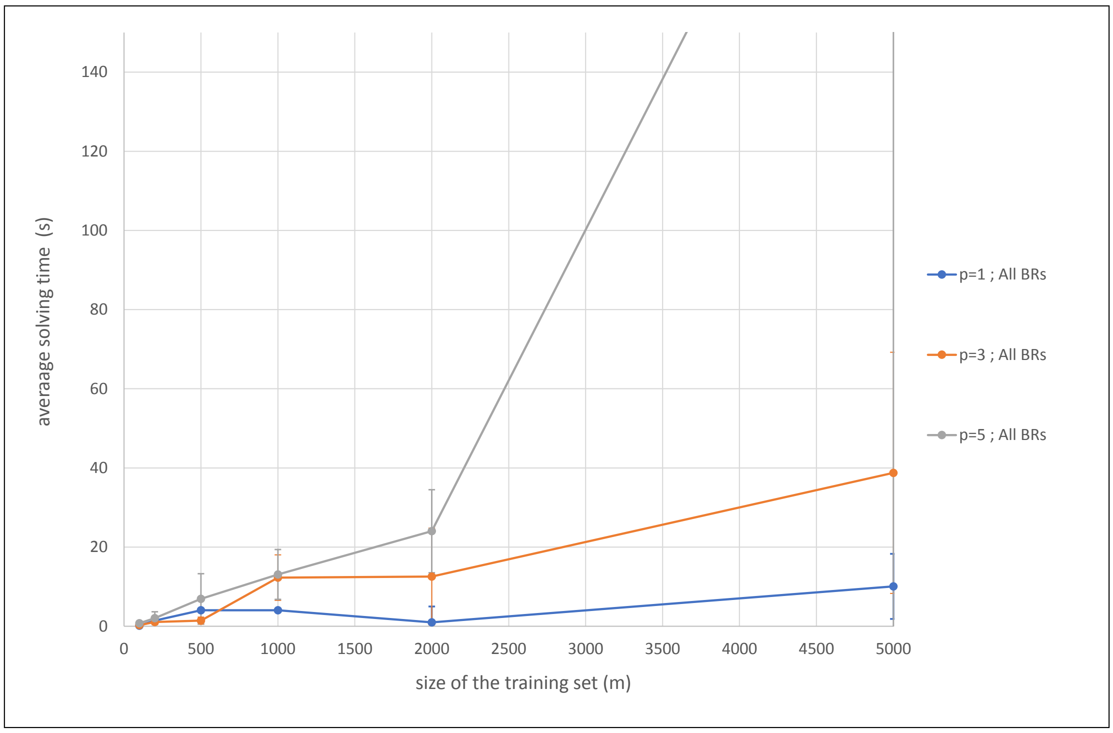


Figure 6.9: **Average solving time** over a range of values of m for 100 randomly generated BRs with error bars representing the standard deviation

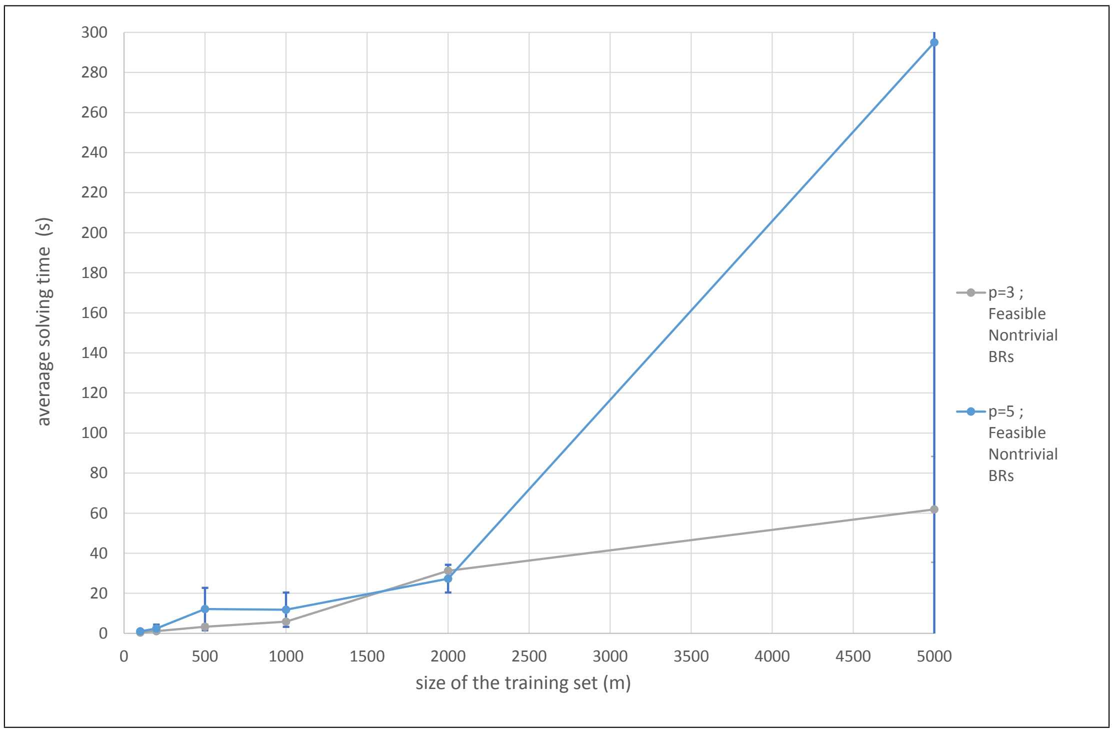


Figure 6.10: **Average solving time** over a range of values of m for the feasible, nontrivial BRs among 100 randomly generated BRs with error bars representing the standard deviation

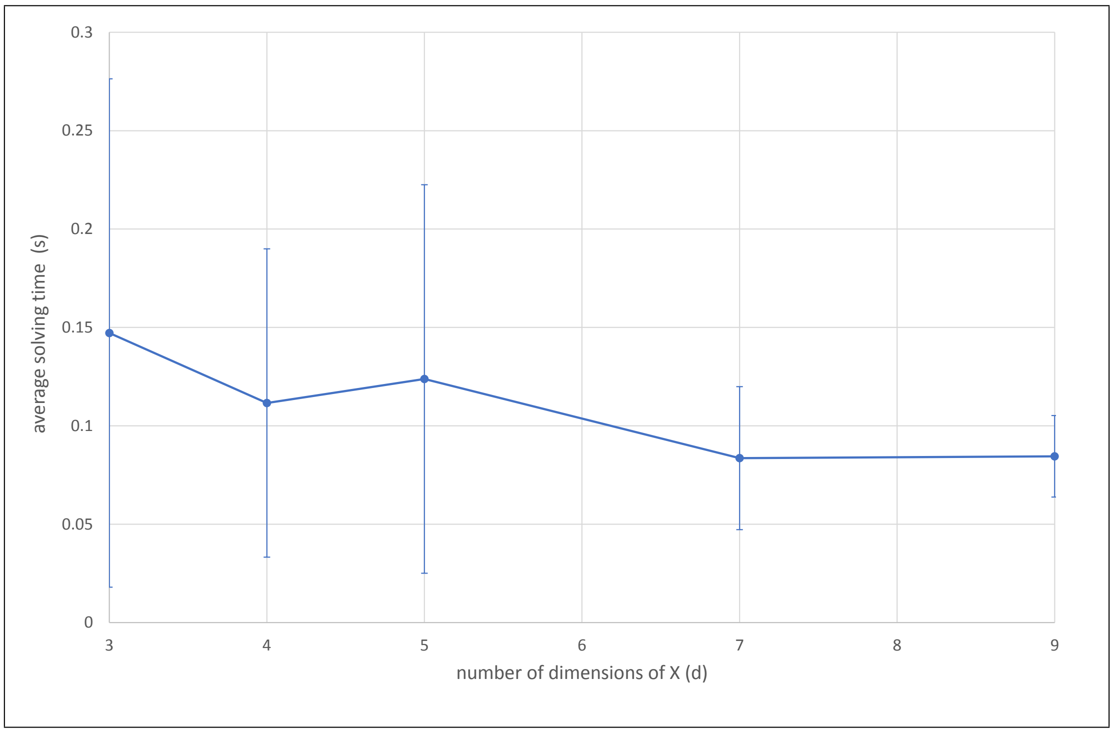


Figure 6.11: **Average solving time** over a range of values of d

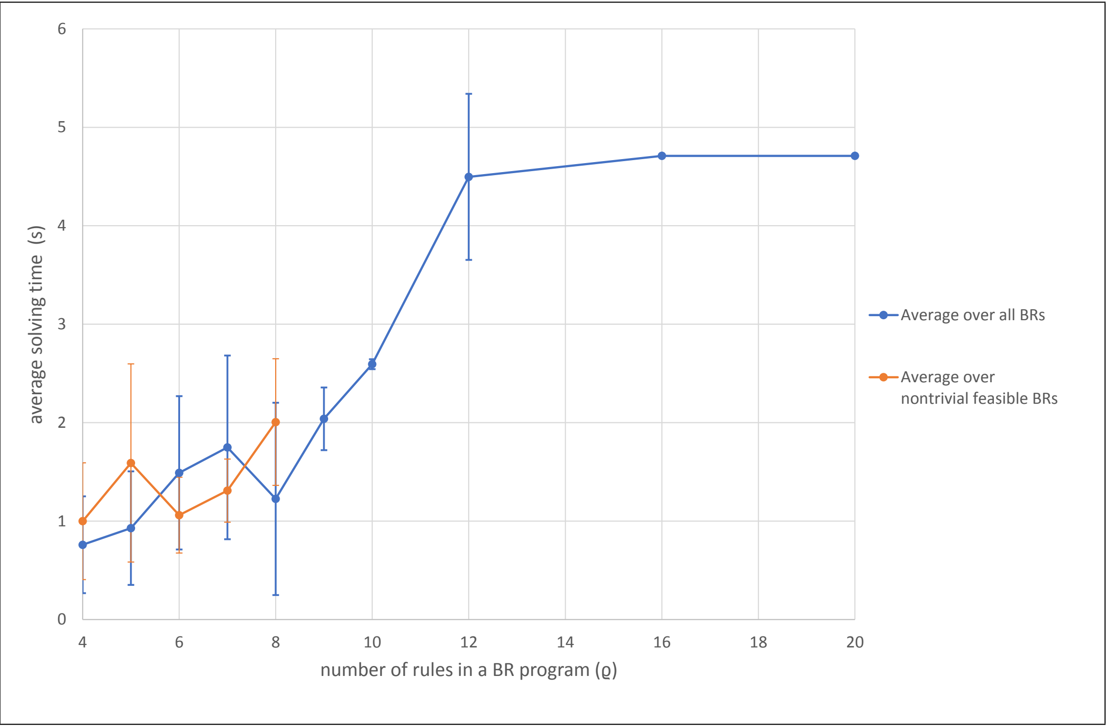
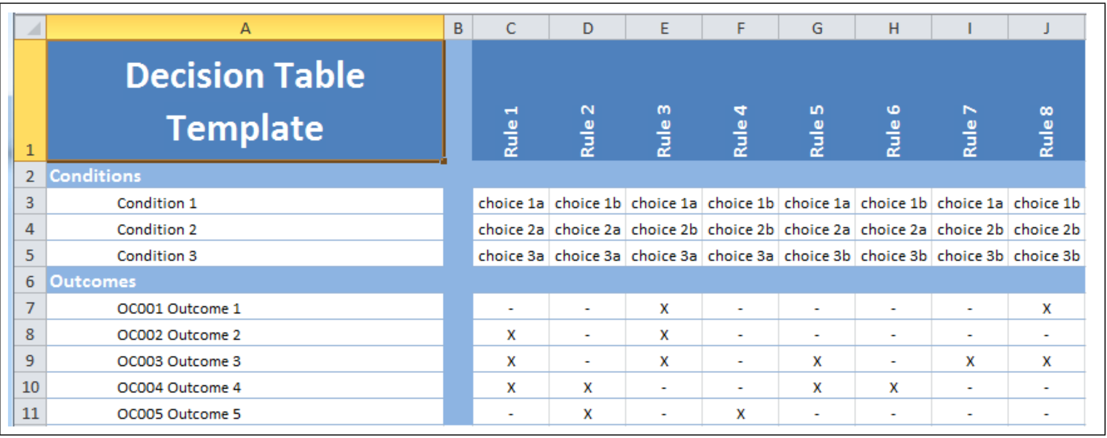


Figure 6.12: **Average solving time** over a range of values of ρ



	A	B	C	D	E	F	G	H	I	J
1	Decision Table Template		Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
2	Conditions									
3	Condition 1		choice 1a	choice 1b	choice 1a	choice 1b	choice 1a	choice 1b	choice 1a	choice 1b
4	Condition 2		choice 2a	choice 2a	choice 2b	choice 2b	choice 2a	choice 2a	choice 2b	choice 2b
5	Condition 3		choice 3a	choice 3a	choice 3a	choice 3a	choice 3b	choice 3b	choice 3b	choice 3b
6	Outcomes									
7	OC001 Outcome 1		-	-	X	-	-	-	-	X
8	OC002 Outcome 2		X	-	X	-	-	-	-	-
9	OC003 Outcome 3		X	-	X	-	X	-	X	X
10	OC004 Outcome 4		X	X	-	-	X	X	-	-
11	OC005 Outcome 5		-	X	-	X	-	-	-	-

Figure 6.13: A decision table template (source: http://www.seilevel.com)

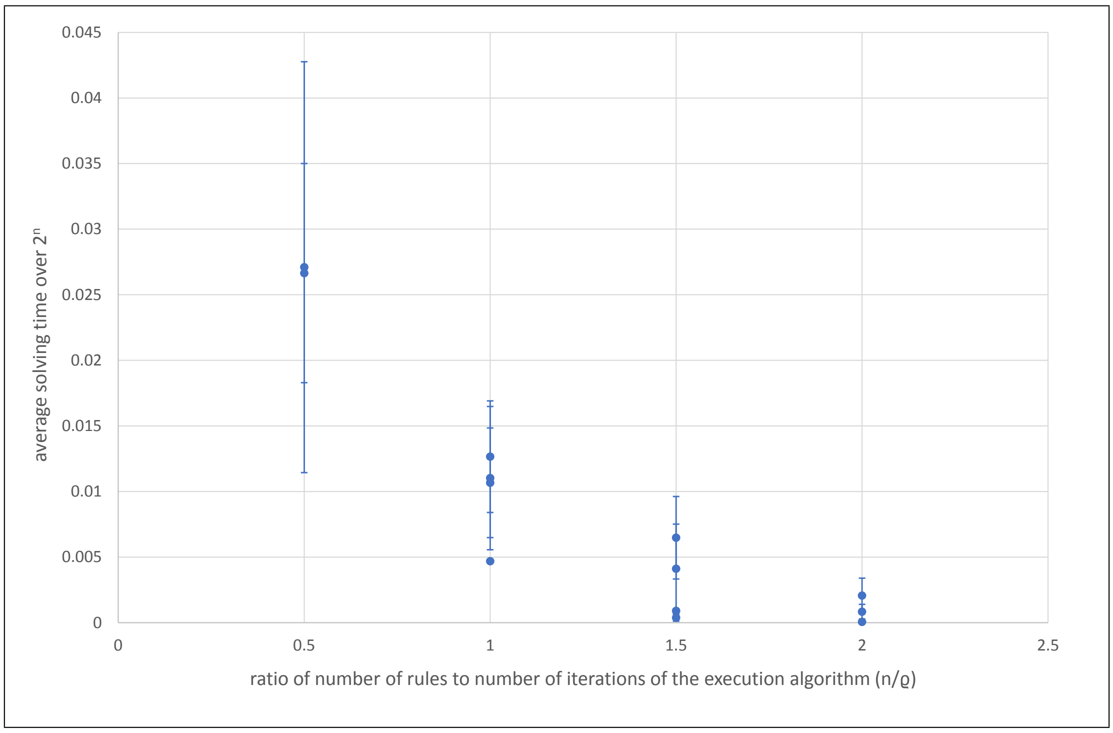


Figure 6.14: **Average solving time** divided by 2^n over a range of values of n/ρ

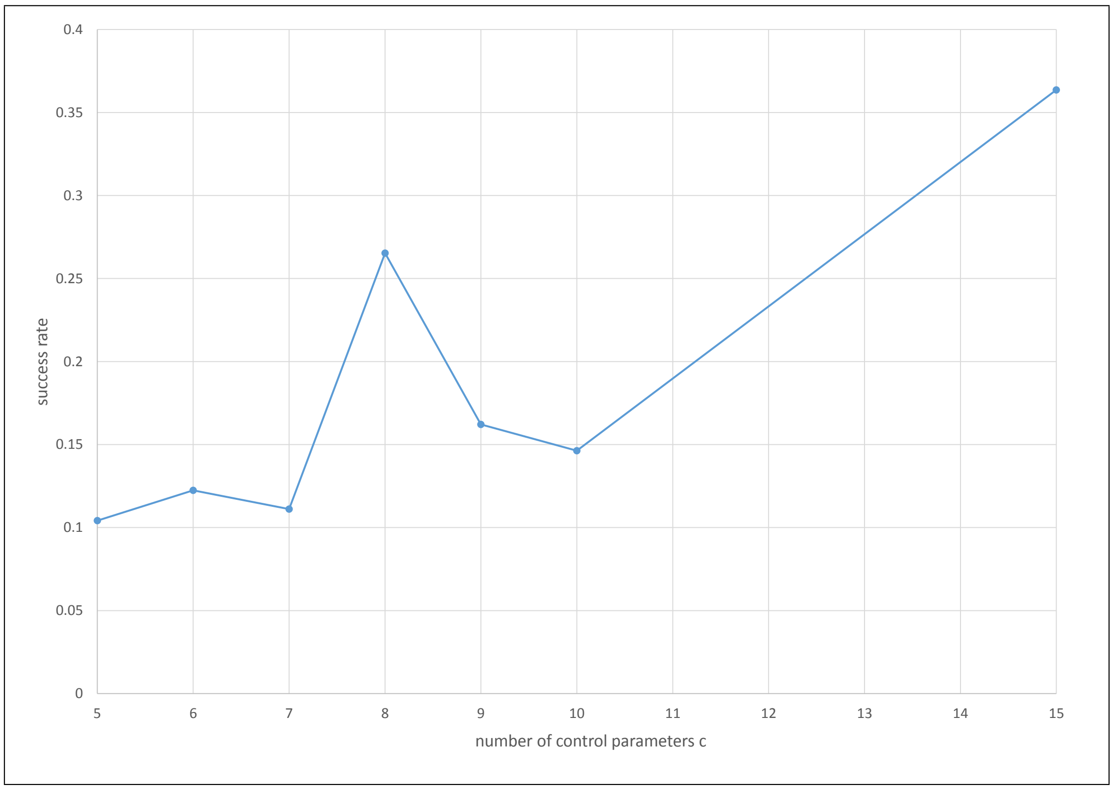


Figure 6.15: **Success rate** over non-trivial solvable P_1 for varying values of c in seconds

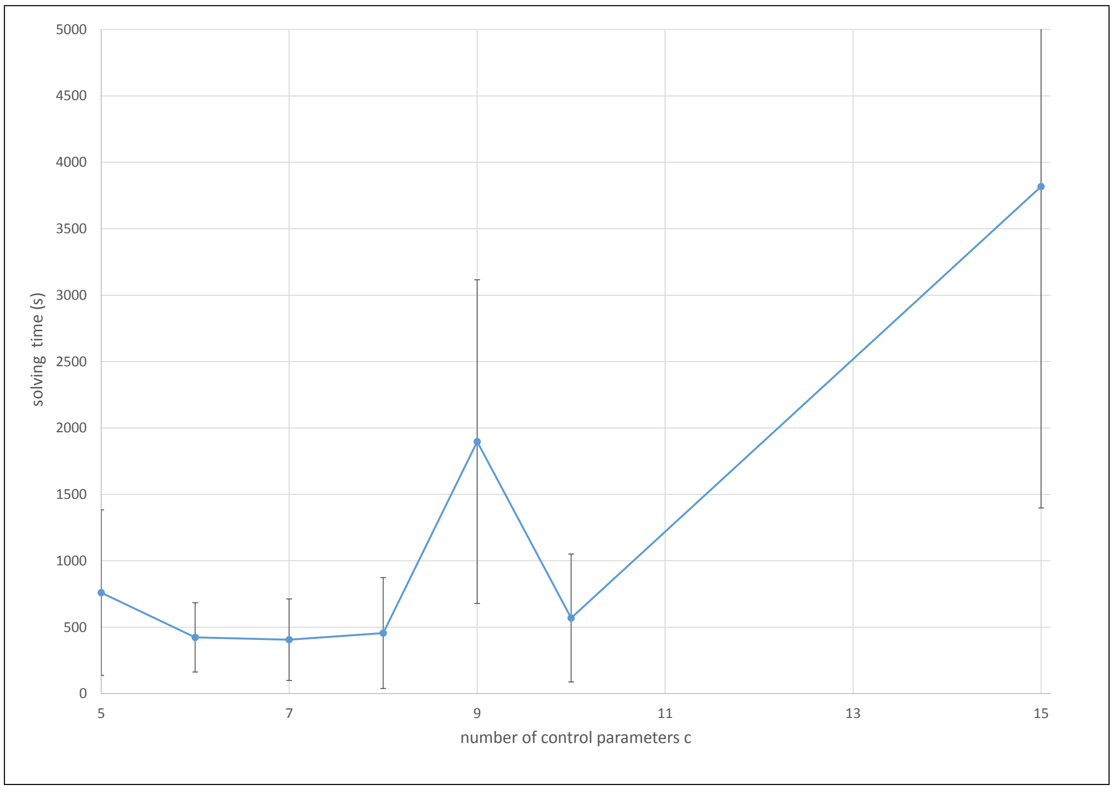


Figure 6.16: **Average solving time** over non-trivial solvable P_1 for varying values of c in seconds

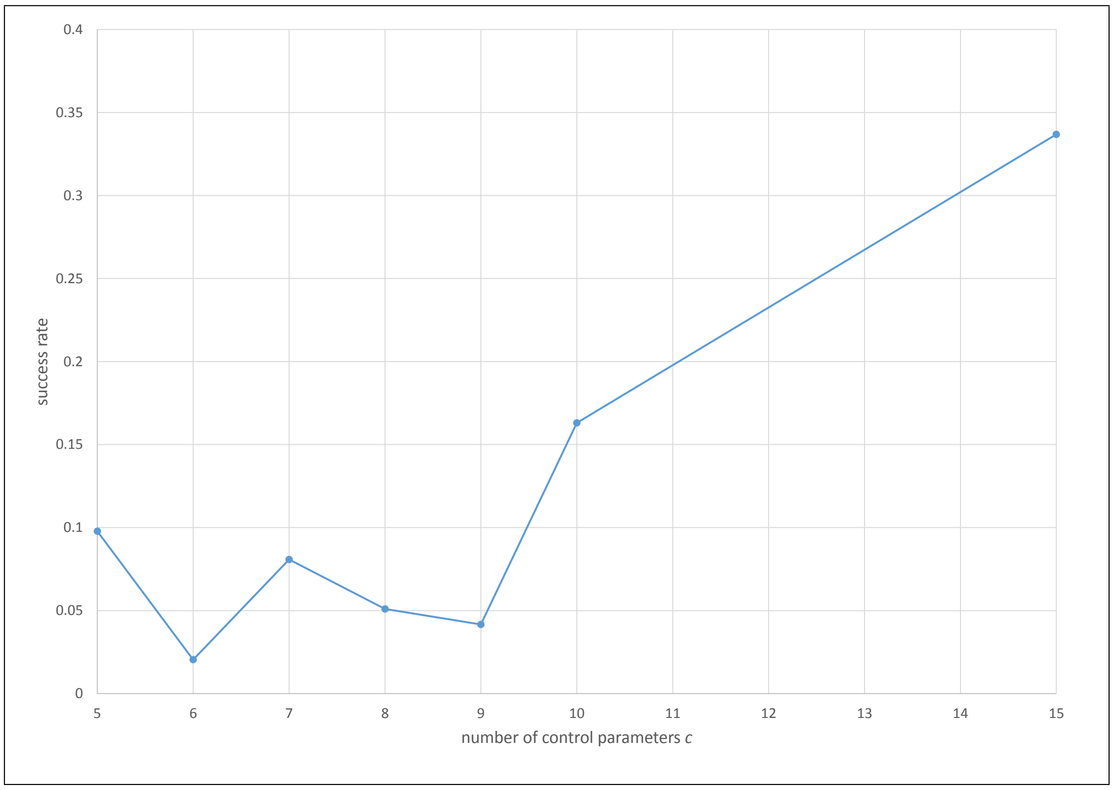


Figure 6.17: **Success rate** over non-trivial solvable P_2 for varying values of c in seconds

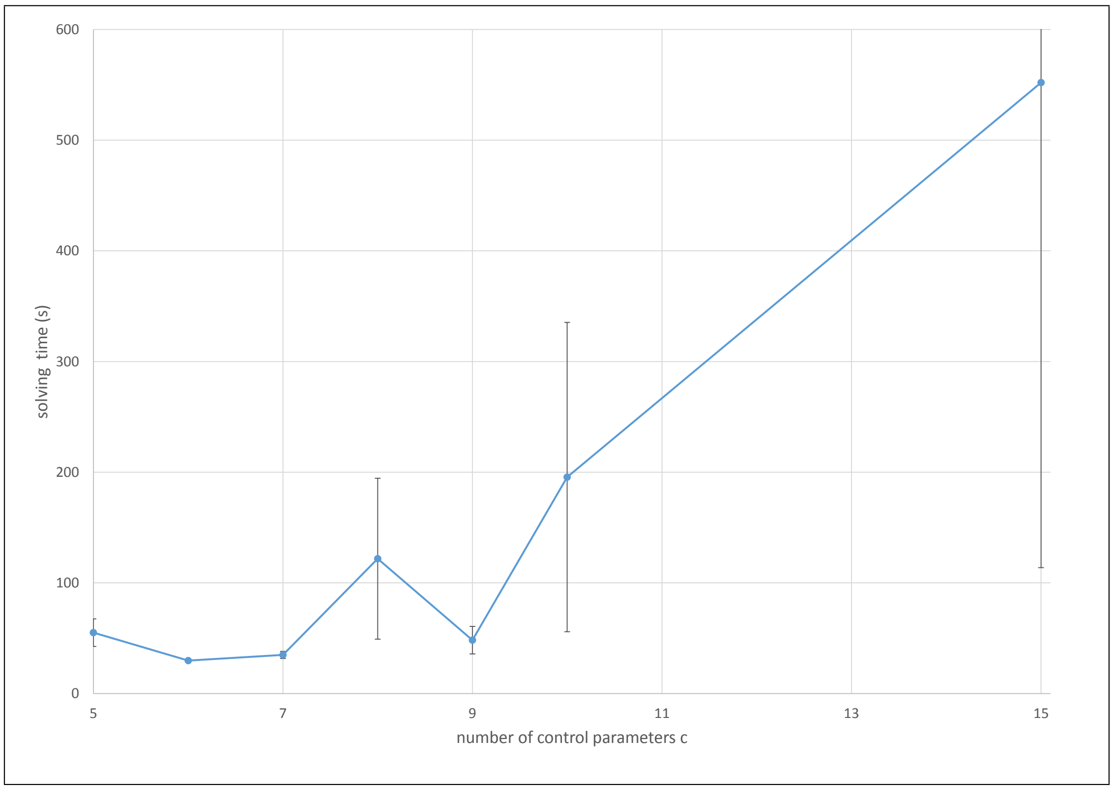


Figure 6.18: **Average solving time** over non-trivial solvable P_2 for varying values of c in seconds

Chapter 7

Discussion

In this thesis we have searched for a ML algorithm which can solve the statistical goal learning problem in Eq. 2.2 for BR programs. After having introduced relevant concepts and formalizations relating to both ML and BRs, we have first explored BRs as a programming language, proving that no such algorithm exists in the general case. We then have looked at one of the most common sub-classes of BR programs, iteration bounded BR programs, and outlined an algorithm which can solve the statistical goal learning problem in theory. We refined our work by making the algorithm easier to solve in practice, at the cost of further restraining the type of BR programs we can learn to linear BRs. We have then explored some related learning problems, before providing some experimental data.

In this chapter, we first discuss the value of our contributions in the area of BR research, then we compare our algorithm to existing methods for solving the same problem. We also discuss the value of our thesis from the point of view of BRMS developers such as IBM. Finally, we conclude this thesis with some possible avenues of further research.

7.1 A formal study of BRs

One of the most flagrant gaps in existing research about BRs is the lack of theoretical study of BRs as a programming language. While the semantics and performance of BRMS has been tested [137, 90, 20], and the business modeling aspect of Production Rules has been formalized [108], we have not found any compilation of interpreters or comparison of expressive power.

In the process of our work, we have syntactically defined the BR language and provided two different interpreters which make that language Turing-complete. While modern BRMS implement both the standard execution algorithm $\mathscr{I}_S$ (often wrongly called the 'Rete' algorithm) and so-called 'sequential' algorithms, the latter in fact correspond to degenerate interpreters with an upper bound on the number of iterations of the execution loop. The existing work done on analyzing this non-standard semantics did not compare the expressive power of these languages, or even consider them as distinct programming languages [15].

Furthermore, we have linked the Operational Semantics of the BR language to the standard SOS used in most imperative programming languages [98]. We have shown that this can be used to automatically decide termination of a BR program for given ranges of inputs in some cases. Other Operational Semantics for BR programs exist [137, 16], however they are always considered on their own and require specific analysis tools that must be developed for each BRMS. In particular, a use of our SOS that they cannot achieve directly is the automatic comparison or even translation of BR programs written in a BR language variant (given a different syntax or interpreter) to another.

We have provided an explanation for the fact that most BRs use typed meta-variables, as well as proved that they do not influence the expressive power of BRs. This makes formal study of the BR language much easier, however we have not quantified the loss of computational performance due to not using typed meta-variables. Further research seeking to examine how BR sets defined in BRMS behave as BR programs could pursue this avenue.

7.2 New learning algorithms

We have introduced a new learning problem which is prominent in business interests but has not been explored much in the ML research community. The "narrow" version of the statistical goal learning problem we examined in detail can be treated as a straighforward black-box or derivative-free optimization problem [114, 38], when written in the following form:

$$\left.\begin{array}{rcl}\min_p \left|\mathbb{E}_{q\in Q}\Big[f\Big(P(p,q)\Big)\Big]\right| & & \\ \|p-p^0\| & \leq & \varepsilon,\end{array}\right\} \tag{7.1}$$

Not only is this form much less useful to industrial applications of BR programs, the properties of the function $\mathbb{E}_{q\in Q}(P(.,q))$ are not always clear in the case where P is a BR program. This makes relying on established methods very hit-or-miss and as the cost of the simulation is quite high for industrial BR programs, we provide an alternative algorithm adapted for specific classes of BR programs.

The more general statistical goal learning problems do not always correspond to the basic variation of the usual black-box optimization problem, in which only the objective function is black-boxed. For example, the case where we impose a maximum percentage of a given output only corresponds to a black-box optimization problem where the objective is known, and it is the constraint $\mathbb{E}_{q\in Q}\Big[P(p,q)\Big] \leq g$ which is a black-box [13].

Another possible approach to the same problem is that of trying to learn a statistical distribution. Given $\mathscr{D}(p)$ the statistical distribution of the values

$P(p, q), q \in Q$, we can use the following formulation:

$$\left.\begin{array}{rcl} \min\limits_{d \in \mathscr{D}(\pi)} \|d - \mathscr{D}(p^0)\| & & \\ f(d) & \leq & 0, \end{array}\right\} \tag{7.2}$$

where f is a function of statistical distributions over Q and π is the set of possible values for the parameter p. The equivalence between the distance over distributions and the distance over π used in Eq. 2.2 is not always true, however both can be equally useful in application cases. By relaxing this approach to having d be any distribution of $\mathscr{R}^Q$ (assuming P has scalar output), the difficulty of simulating the BR program becomes the difficulty of reverse computing p from d, which depending on the BR can be easier or harder than the initial problem.

7.3 Industrial applications

A very relevant result of our thesis is that of the theoretical feasibility of the original problem, which has great industrial relevance, notably to IBM and other BRMS vendors in terms of answering their client's needs. Furthermore, the proof of concept work we have done demonstrates how a statistical goal learning problem can be solved in an entirely automated way, from parsing the set of BRs to getting a new value of the parameter. The last step towards true automation, reinjecting the new value of the parameter in the ruleset, can also be automated – although it probably should not, as human control is often required before modifying the ruleset.

The main limitation of our algorithm which makes a direct application to ODM or other BRMS unlikely is the lack of scalability. While our data indicates a severe lack in this respect, the MILP problem associated with specific rulesets that need to be learned regularly can always be studied specifically – using reformulations or heuristics linked to properties of the ruleset. Improvements to the general performance of the learning algorithm may be found either through reformulation or by further limiting the class of BRs we consider. A notable reformulation which may have a very variable effect depending on the BR program is to start the simulation at the first occurrence of a rule being executed with a priority equal or higher than that of the rule in which the parameter p is relevant. This reduction in size would depend on the input Q and be difficult to automate, as it would remove an unequal number of $x^{1,j}, \ldots, x^{\text{start},j}$ for each j.

Relevant existing heuristics may also be used to find at least feasible solutions. Fulfilling the requested constraint on the BR program without minimizing the distance to the original parameters in this way may be more relevant to BRMS users, if fast enough, than a more exact solution. Other custom heuristics, e.g. taking advantage of the fact that many values are fixed until the first occurrence of p in the BR execution, may also help. The infeasibility of the statistical learning problem Eq. 2.2 appearing in Sec. 2.2 is also a relevant information to BRMS users: assuming the tolerance ε is well-chosen (or imposed externally), infeasibility of the MILP problem produced by our algorithm indicates that the structure or the ruleset prevents the stated goal. The recommended action in this case is to modify the BR program by removing (or in some cases adding) a BR.

My intuition is that our algorithm can be useable in commercial BRMS given some work over the aforementioned aspects. This is because we can reasonably suppose that industrial BR programs are likely to be easier to learn on average than randomly generated BR programs, and in my opinion are in fact much easier to learn than such BR programs. An indication that industrial problems might be solvable is the fact that most industrial business processes do not use complex rules, but rather many rules each applying to one or two components of the variable x. This corresponds to using sparse matrices for L, G, A and B, which appropriate solvers can use to their advantage. Furthermore, the scalability needed is not as high as it appears at first glance, as industrial programs can often be preprocessed by removing some irrelevant rules, e.g. where the actions are side effects (the affected variables are not the output and are never tested); and elements of the training set, e.g. where the execution algorithm terminates without ever having the opportunity to execute a rule where a parameter appears. As for rules in which no parameter appears, I do not believe they can be removed unobtrusively: the truth value of their condition is always dependent on the current value of the variables, which is never guaranteed to be independant from the parameters p.

7.4 Conclusion

The original question asked by users of IBM ODM about the possibility and practicality of using ML to satisfy statistical goals when parametrizing BR programs has been answered in a few different ways. The general case has proved to be impossible to learn. We have provided an algorithm for the practical cases of Linear Iteration Bounded BR programs as well as an idea of the restrictions over any industrial application. The work needed to improve scalability is the improvement on the computational complexity of solving the MILP problem associated with our algorithm, with the most interesting avenue being finding a custom heuristics.

In the process, we have also formally proven a commonly accepted but never proven properties of BR programs: they are Turing-complete. This comes with a Sequential Operational Semantics which can be used to prove termination of a BR program over an interval of inputs. Further exploiting this SOS to automatically convert a BR program to another language, or even to another BR language (using a different interpreter), is another possible continuation of our work.

Finally, restricting the class of BRs we wish to be learning even further might lead to simplifications in the MILP problem associated with our algorithm. As many BR programs used by businesses are simple variations on Decision Trees, studying those variations and creating algorithms for the statistical learning problem associated with these specific BR programs can be a practical area of research for BRMS developers to be interested in.

We have proved that such algorithms are outright necessary to address the needs of BR users in terms of statistical goal learning, as no general algorithm exists. We have provided a learning algorithm for the problem in Eq. 2.2 in Sec. 2.2, which can in the future be adapted by BRMS developers such as IBM into practical products.

Appendix A

Complete proof of some theorems

This appendix contains the complete proofs for Theorems 4.8, 4.9 and 4.10, which are simple but tedious, and were not included in extenso in the text for that reason. We have instead chosen to give the idea of each proof in the text, and the fully detailed proof in the following Sections.

A.1 Complete proof of Theorem 4.8

In this section, we prove Theorem 4.8 from Subsection 4.4.1. To be precise, we prove that any optimal solution p to the MIP problem from Fig. 4.4 is also a solution to Eq. 2.2, given a LIBBR program of the form described in Section 4.3 and a statistical goal which only considers one relevant output in the form of $x1$, i.e. $f(P(p, q)) = P(p, q)_1$.

As Theorem 4.4 shows that the MP problem in Fig. 4.3 solves Eq. 2.2, a simple transcription of this MP problem using the LIBBR and statistical goal assumptions shows that the MIP problem in Fig. A.1 also does. Consequently, we only have to prove that the MIP problem in Fig. 4.4 is a reformulation of the problem in Fig. A.1.

$$
\begin{array}{lr}
\underset{p,x,y}{\text{minimize}} & \left|p^0 - p\right| \\
\text{subject to} & \\
 & \text{(C2), (C3),(C4), (C5), (IC1)} \\
\forall (i,j) \in I\backslash\{n\}\times J & x^{i+1,j} = \sum_{r\in R} (A_r(p)\, x^{i,j} + B_r(p))\, y_{ijr} \\
 & + (1 - \sum_{r\in R} y_{ijr}) x^{i,j} \\
 & \left|\frac{1}{m}\sum_{j\in J} x_1^{n,j} - g\right| \leq \varepsilon \\
\forall (i,j) \in I\times J & x^{i,j} \in X \\
 & p \in \mathbb{R} \\
\forall (i,j,r) \in I\times J\times R & y_{ijr}, \in \{0,1\}
\end{array}
$$

Figure A.1: **Initial reformulation of the MP problem in Fig. 4.3** under the assumptions of Subsection 4.4.1

Using the assumption that the only parameter p takes the place of $A_{1,1,1}$, we know that the values of $A_r(p)$ are defined as in (IC2), (IC3) and (IC4):

$$
\begin{array}{lrl}
\forall r \neq 1, & A_r(p) & = A_r \\
 & A_1(p)_{1,1} & = A_{1,1,1} \\
\forall (h,k) \neq (1,1), & A_1(p)_{h,k} & = A_{1hk}
\end{array}
$$

and that the values of $B_r(p)$ are simply the same as the original B_r. The first non-numbered constraint in the formulation in Fig. A.1 is thus equivalent to (C1′), (IC2), (IC3) and (IC4). Furthermore, the second non-numbered constraint in that formulation is exactly constraint (C6′) from Fig. 4.4.

We now prove that replacing {(C3), (C4)} by $\{(\text{C3}'), (\text{C4}'_1), (\text{C4}'_2), (\text{C4}'_3)\}$ is a reformulation of the MP problem. In other words, we now prove that the MP problem in Fig. 4.4 and the problem in Fig. A.1 have the same feasible solutions. We proceed by first proving that feasible solution of the first problem are also feasible solution in the second, then proving the reverse.

Lemma A.1. *Feasible solutions of Fig. 4.4 are feasible solutions of Fig. A.1.*

Proof. Suppose p, a, x, y, y^L, y^H a feasible solution for the problem in Fig. 4.4. As discussed above, p, x, y satisfies every constraint in Fig. A.1 except for (C3) and (C4). We proceed by induction over the rule $r \in R$.

For the first rule $\mathcal{R}_1$, satisfying (C3) and (C4) is equivalent to having $y_{ij1} = T_1(x^{i,j})$. In the case $y_{ij1} = 1$, the constraint (C3′) from Fig. 4.4 gives us directly $L_1 \leq x^{i,j} \leq H_1$, i.e. $T_1(x^{i,j}) = 1$. In the case $y_{ij1} = 0$, we have according to (C4′$_3$):

$$\exists k \in D \mid y^H_{ij1k} = 0 \vee y^L_{ij1k} = 0$$

Using either constraint (C4′$_1$) or constraint (C4′$_2$), we have (barring the equality case[1]):

$$\exists k \in D \mid L_{1k} > x^{i,j}_k \vee x^{i,j}_k > H_{1k}$$

which is exactly the definition of $T_1(x^{i,j}) = 0$ for LIBBR.

We now suppose that (C3) and (C4) are satisfied for any $i \in I$, $j \in J$, and $r' < r$, where $r \in R$. We again consider both cases $y_{ijr} = 1$ and $y_{ijr} = 0$, and prove that the constraints are satisfied.

We first consider the case $y_{ijr} = 1$. The equality $L_r \leq x^{i,j} \leq H_r$ is a direct consequence of (C3′), thus we have $T_r(x^{i,j}) = 1$. The constraints (C3) and (C4) are thus trivially satisfied, as $\sum\limits_{r'<r} y_{i,r'}$ is positive.

In the case $y_{ijr} = 0$, (C3) is trivially satisfied. According to constraint (C4′$_3$),

[1]We suppose that the equality case is irrelevant for continuous x variables

we have as in the case $r = 1$:

$$\exists k \in D \mid y^H_{ijrk} = 0 \vee y^L_{ijrk} = 0$$

Using either constraint (C4$'_1$) or constraint (C4$'_2$) as appropriate, we thus have:

$$\exists k \in D \mid x^{i,j}_k \geq H_{rk} \vee x^{i,j}_k \leq L_{rk} \vee \sum_{r'<r} y_{ijr'} = 1$$

Remarking that the first parts of this Boolean conjunction can be expressed in terms of $T_r(x^{i,j})$, we have in fact $T_r(x^{i,j}) = 0 \vee \sum_{r'<r} y_{ijr'} = 1$. Consequently, we have $0 \geq T_r(x^{i,j}) - \sum_{r'<r} y_{ijr'}$, which ensures that (C4) is satisfied.

The induction thus proves that the constraints (C3) and (C4) are satisfied for all $r \in R$. Since p, x, y satisfies every constraint in Fig. A.1, it is indeed a feasible solution for that problem. □

Lemma A.2. *Feasible solutions of Fig. A.1 are also feasible solutions of Fig. 4.4.*

Proof. Suppose p, x, y a feasible solution for the problem in Fig. A.1. As discussed above, it satisfies every constraint in Fig. 4.4 except for (C3$'$), (C4$'_1$), (C4$'_2$), and (C4$'_3$). For any $i \in I$, $j \in J$, $r \in R$, we now consider the two cases $y_{ijr} = 1$ and $y_{ijr} = 0$.

In the case where $y_{ijr} = 1$, we have $T_r(x^{i,j}) = 1$, i.e. $L_r \leq x^{i,j} \leq H_r$. Constraint (C3$'$) is thus satisfied. Regardless of which value is taken by y^H and y^L, constraint (C4$'_3$) is also satisfied. A choice of $y^H_{ijrk} = y^L_{ijrk} = 1$ for all $k \in D$ suffices for (C4$'_1$) and (C4$'_2$) to also be satisfied.

In the case where $y_{ijr} = 0$, the constraint (C3$'$) is always satisfied. Furthermore, constraint (C4) of Fig. A.1 gives us $T_r(x^{i,j}) \leq \sum_{r'<r} y_{ijr'}$. This can also be expressed as:

$$T_r(x^{i,j}) = 0 \vee \sum_{r'<r} y_{ijr'} = 1$$

We consider separately the case where each clause of this conjunction is true.

In the first case, we have:

$$\exists k \in D \mid L_{rk} > x_k^{i,j} \vee x_k^{i,j} > H_{rk}$$

We note $h \in D$ satisfying this property. As (C4$'_1$) and (C4$'_2$) are symmetrical, we assume without loss of generality that $L_{rh} > x_h^{i,j}$. Assigning the values $\forall k \in D, y^H_{ijrk} = 1$, $\forall k \neq h, y^L_{ijrk} = 1$ and $y^L_{ijrh} = 0$ satisfies the constraints (C4$'_1$) and (C4$'_2$) for all values of k. Furthermore, since $\sum\limits_{k\in D}(y^H_{ijrk} + y^L_{ijrk}) = 2d - 1$, this solution also satisfies (C4$'_3$).

In the second case, we have $\sum\limits_{r'<r} y_{ijr'} = 1$, which automatically satisfies both constraint (C4$'_1$) and constraint (C4$'_2$) for every $k \in D$. We can thus choose $y^H_{ijrk} = y^L_{ijrk} = 0$ and also satisfy (C4$'_3$).

Using the constructed values for y^L and y^H, the solution p, a, x, y, y^L, y^H satisfies every constraint in Fig. 4.4, it is thus a feasible solution. Note that there is no chance of our construction for y^L and y^H hitting a contradiction, since we have only ever chosen values once for each (i, j, r, k) tuple. □

We have proven that the MIP problem in Fig. 4.4 is a reformulation of a problem known to solve Eq. 2.2, which proves Theorem 4.8. □

A.2 Complete proof of Theorem 4.9

In this section, we prove Theorem 4.9 from Subsection 4.4.1. To be precise, we prove that any optimal solution p of the MILP in Fig. 4.5 is is also a solution to Eq. 2.2, given a LIBBR program of the form described in Section 4.3, a statistical goal which only considers one relevant output in the form of $x1$, i.e. $f(P(p, q)) = P(p, q)_1$ and a parameter p which takes the place of $B_{1,1}$.

As was proven above in Section A.1, the MIP program in Fig. 4.4 solves Eq. 2.2 when the parameter takes the place of $A_{1,1,1}$. Simply changing which element of the BR program is a parameter provides the MIP program in Fig. A.2, which thus solves Eq. 2.2 when the parameter takes the place of $B_{1,1}$. Consequently, we only have to prove that the MILP problem in Fig. 4.5 is a reformulation of the problem in Fig. A.2.

$$
\boxed{
\begin{array}{lcr}
\underset{p,a,x,y,y^H,y^L}{\text{minimize}} & \left|p^0 - p\right| & \\
\text{subject to} & & \\
 & (\mathrm{C1}'), (\mathrm{C2}), (\mathrm{C3}'),(\mathrm{C4}'_1), (\mathrm{C4}'_2), (\mathrm{C4}'_3) & \\
 & (\mathrm{C5}), (\mathrm{C6}')(\mathrm{IC1}) & \\
\forall r \in \{2, \ldots, \rho\}, & b_r = B_r & (\mathrm{IC2}') \\
 & b_{1,1} = p & (\mathrm{IC3}') \\
\forall k \in \{1, \ldots, d\}, & b_{1k} = B_{1k} & (\mathrm{IC4}') \\
\forall (i,j) \in I \times J & x^{i,j} \in X & \\
\forall k \in R & a_k \in \mathbb{R}^{d\times d} & \\
 & p \in \mathbb{R} & \\
\forall (i,j,r,k) \in I \times J \times R \times D & y_{ijr}, y^H_{ijrk}, y^L_{ijrk} \in \{0,1\} &
\end{array}
}
$$

Figure A.2: **MIP Formulation with p Taking the Place of B_{11}** with $e = (1, \ldots, 1) \in \mathbb{R}^d$ a vector of all ones

Constraint (C6′) is trivially equivalent to its linearized form (C6″). We now prove that replacing (C1′) by {(C1″$_1$), (C1″$_2$), (C1″$_3$)} is a reformulation of the MP problem. To do so, we simply prove that for any feasible solution p, b, x, y, y^L, y^H, w of Fig. 4.5, we have:

$$\forall i \in I, j \in J, r \in R, w_{ijr} = (A_r x^{i,j} + b_r) y_{ijr} \tag{A.1}$$

In such a feasible solution, let us consider the two cases where $y_{ijr} = 1$ and

$y_{ijr} = 0$. In the first case, we have thanks to (C1$''_3$):

$$A_r x^{i,j} + b_r - x^{i,j} \leq w_{ijr} \leq A_r x^{i,j} + b_r - x^{i,j}$$

which gives us directly Eq. A.1. In the second case, we use (C1$''_2$) to get the value of the variable we need: $w_{ijr} = 0$. This is also the value which matches Eq. A.1.

As the only difference between (C1$''_1$) in Fig. 4.5 and (C1$'$) in Fig. A.2 is replacing w with the value expressed in Eq. A.1, this both proves that any feasible solution p, b, x, y, y^L, y^H, w of Fig. 4.5 is also a feasible solution to Fig. A.2. Any feasible solution to the latter is also a feasible solution to the first by choosing the value of w to be $w_{ijr} = (A_r x^{i,j} + b_r) y_{ijr}$.

As the MIP problem in Fig. A.2 solves Eq. 2.2, the MILP problem in Fig. 4.5 also does, as its reformulation. □

A.3 Complete proof of Theorem 4.10

In this section, we prove Theorem 4.10 from Subsection 4.4.1. To be precise, we prove that any optimal solution p of the MILP in Fig. 4.6 is is also a solution to Eq. 2.2, given a LIBBR program of the form described in Section 4.3, a statistical goal which only considers one relevant output in the form of $x1$, i.e. $f(P(p, q)) = P(p, q)_1$ and a parameter p which takes the place of $A_{1,1,1}$.

As was proven above in Section A.1, the MIP program in Fig. 4.4 from Subsection 4.4.1 solves Eq. 2.2. Consequently, we only have to prove that the MILP problem in Fig. 4.6 is a reformulation of the problem in Fig. 4.4.

Constraint (C6$'$) is trivially equivalent to its linearized form (C6$''$). We now prove that replacing (C1$'$) by {(C1$''_1$), (C1$''_2$), (C1$'''_3$), (C1$'''_4$), (C1$'''_5$)} is a reformulation of the MP problem. To do so, we simply prove that for any feasible solution

p, b, x, y, y^L, y^H, w of Fig. 4.5, we have:

$$\begin{aligned} \forall i \in I, j \in J, r \in R, \quad (k,h) \in D^2, z_{ijrhk} &= za_{rhk}x_k^{i,j} \\ \forall i \in I, j \in J, r \in R, \quad (k,h) \in D^2, z_{ijrhk} &= a_{rhk}x_k^{i,j} \end{aligned} \tag{A.2}$$

which we then use to prove Eq. A.1 is still true:

$$\forall i \in I, j \in J, r \in R, w_{ijr} = (a_r x^{i,j} + B_r) y_{ijr} \tag{A.3}$$

In such a feasible solution, we first consider the two cases where $a_{rhk} = 1$ and $a_{rhk} = 0$. In the first case, we have thanks to ($\text{C1}_5'''$):

$$z_{ijrhk} = x_k^{i,j} = a_{rhk}x_k^{i,j}$$

which gives us directly Eq. A.2. In the second case, we use ($\text{C1}_4'''$) to get the value of the variable we need: $z_{ijrhk} = 0$. This is also the value which matches Eq. A.2.

Given Eq. A.2, the matrix product $a_r x^{i,j}$ is thus expressed:

$$a_r x^{i,j} = \sum_{h \in D} a_{rhk}x_k^{i,j} = \sum_{h \in D} z_{ijrhk}$$

When considering the two cases where $y_{ijr} = 1$ and $y_{ijr} = 0$, we use this to prove Eq. A.1. In the case $y_{ijr} = 1$, we have thanks to ($\text{C1}_3'''$):

$$a_r x^{i,j} + B_r - x^{i,j} \leq w_{ijr} \leq a_r x^{i,j} + B_r - x^{i,j}$$

which gives us directly Eq. A.1. In the second case, we use ($\text{C1}_2''$) to get the value of the variable we need: $w_{ijr} = 0$. This is also the value which matches Eq. A.1.

As the only difference between ($\text{C1}_1''$) in Fig. 4.6 and ($\text{C1}'$) in Fig. 4.4 is replacing w with the value expressed in Eq. A.1, this both proves that any feasible solution

$p, b, x, y, y^L, y^H, w, z$ of Fig. 4.6 is also a feasible solution to Fig. 4.4. Any feasible solution to the latter is also a feasible solution to the first by choosing the values of z and w to be:

$$\begin{aligned} z_{ijrhk} &= a_{ijrhk} x_k^{i,j} \\ w_{ijr} &= (\sum_{k \in D} z_{ijrhk} + b_r) y_{ijr} \end{aligned}$$

As the MIP problem in Fig. 4.4 solves Eq. 2.2 from Th. 4.8, the MILP problem in Fig. 4.6 also does, as its reformulation. $\square$

Bibliography

[1] S. Abiteboul and V. Vianu.
"Datalog Extensions for Database Queries and Updates".
In: *Journal of Computer and System Sciences* 43.1 (1991), pp. 62–124.

[2] H. Agli et al. "Business Rules Uncertainty Management with Probabilistic Relational Models".
In: *Rule Technologies. Research, Tools, and Applications.*
Ed. by J.J. Alferes et al. 10th International RuleML Symposium (RuleML 2016). Berlin, Germany: Springer, 2016, pp. 53–67.

[3] R. Agrawal and R. Srikant.
"Fast Algorithms for Mining Association Rules".
In: *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)* (1994), pp. 487–499.

[4] D.O. Aihe and A.J. Gonzalez.
"Correcting flawed expert knowledge through reinforcement learning".
In: *Expert Systems with Applications* 42.17-18 (2015), pp. 6457–6471.

[5] E. M. Gold. "Language Identification in the Limit".
In: *Information and Control* 10.5 (1967), pp. 447–474.

[6] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Pacific Grove, CA: Duxbury Press, 2002.

[7] R. Misener and C.A. Floudas. "ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations". In: *Journal of Global Optimization* 59.2 (2014), pp. 503–526.

[8] G. Antoniou et al. *Combining rules and ontologies: a survey.* Technical Report. `http://rewerse.net/publications/rewerse-description/REWERSE-DEL-2005-I3-D3.htm/`. Linköping University, 2005.

[9] Y. Anzai and H.A. Simon. "The theory of learning by doing". In: *Psychological Review* 86.2 (1979), pp. 124–140.

[10] K. R. Apt. *Principles of Constraint Programming.* Cambridge: Cambridge University Press, 2003.

[11] B. Kamsu-Foguem, F. Rigal, and F. Mauget. "Mining association rules for the quality improvement of the production process". In: *Expert systems with applications* 40.4 (2013), pp. 1034–1045.

[12] N.V. Sahinidis. "BARON: A general purpose global optimization software package". In: *Journal of Global Optimization* 8.2 (1996), pp. 201–205.

[13] R.B. Gramacy et al. "Modeling an Augmented Lagrangian for Blackbox Constrained Optimization". In: *Technometrics* 58.1 (2016), pp. 1–11.

[14] J. O. Berger. *Statistical Decision Theory and Bayesian Analysis.* New York, NY: Springer-Verlag, 1985.

[15] B. Berstel-Da Silva. “Formalizing Both Refraction-Based and Sequential Executions of Production Rule Programs”.
In: *Rules on the Web: Research and Applications. Proceedings of the 6th International RuleML Symposium (RuleML 2012)* (2012), pp. 47–61.

[16] B. Berstel-Da Silva. “Operational Semantics of Rule Programs”.
In: *Verification of Business Rules Programs.*
Berlin, Germany: Springer, 2013, pp. 63–89.

[17] *Bonmin (Basic Open-source Nonlinear Mixed INteger programming).*
Product Description. `https://projects.coin-or.org/Bonmin/`.

[18] A. Kolber et al. *Defining Business Rules - What are They Really?*
Project Report 3. The Business Rules Group, 2000.

[19] *The Business Rules Manifesto*. Manifesto.
The Business Rules Group, 2002.

[20] L. Rosa et al. “A Comparative Study of Correlation Engines for Security Event Management”. In: *Proceedings of the 10th International Conference on Cyber Warfare and Security (ICCWS 2015).*
Ed. by J. Zaaiman and L. Leenen.
England, UK: Academic Conferences and Publishing International, 2015.

[21] B.G. Buchanan and E.H. Shortliffe, eds. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley Series in Artificial Intelligence).*
Boston, MA: Addison-Wesley Longman Publishing Co., 1984.

[22] R. M. Burstall. “Proving properties of programs by structural induction”.
In: *The Computer Journal* (1969).

[23] J. R. Quinlan. *C4.5: Programs for Machine Learning.*
San Francisco, CA: Morgan Kaufmann Publishers, 1992.

[24] *Mosek ApS*. Product Description.
`https://www.mosek.com/products/mosek/`.

[25] *Coin-or Branch and Cut*. Product Description.
`https://projects.coin-or.org/Cbc/`.

[26] L.M. Berliner. “Statistics, Probability and Chaos”.
In: *Statistical Science* (1992), pp. 69–90.

[27] Thom Frühwirth. “Theory and practice of constraint handling rules”.
In: *Journal of Logic Programming* 37.1-3 (1998), pp. 95–138.

[28] A. Church. “An Unsolvable Problem of Elementary Number Theory”.
In: *American Journal of Mathematics* 58 (1936), pp. 345–363.

[29] W.J. Clancey. “The Epistemology of a Rule-Based Expert System: a Framework for Explanation”.
In: *Artificial Intelligence* 20.3 (1983), pp. 215–251.

[30] C. Culbert and G. Riley. *CLIPS Basic Programming Guide*. 2003.

[31] P.M. Murphy and M.J. Pazzani.
“Revision of Production System Rule-Bases”.
In: *Proceedings of the 11th International Conference on Machine Learning (ICML 1994)* (1994), pp. 199–207.

[32] J.-B. Hiriart-Urruty and C. Lemaréchal.
Fundamentals of Convex Analysis. New York, NY: Springer, 2001.

[33] P. Belotti et al.
“Branching and bounds tightening techniques for non-convex MINLP”.
In: *Optimization Methods and Software* 24.4-5 (2009), pp. 597–634.

[34] A. Cohen, S. Goldwasser, and V. Vaikuntanathan. "Aggregate Pseudorandom Functions and Connections to Learning". In: *Theory of Cryptography* (2015), pp. 61–89.

[35] M.W. Curtis. "A Turing Machine Simulator". In: *Journal of the Association for Computing Machinery* 12.1 (1965), pp. 1–13.

[36] H. Gallaire and J. Minker, eds. *Logic and Data Bases.* New York, NY: Plenum Press, 1978.

[37] R. Davis, B. Buchanan, and E. Shortliffe. "Production Rules as a Representation for a Knowledge-Based Consultation Program". In: *Artificial Intelligence* 8.1 (1977), pp. 15–45.

[38] L.M. Rios and N.V. Sahinidis. "Derivative-free optimization: a review of algorithms and comparison of software implementations". In: *Journal of Global Optimization* 56.3 (2013), pp. 1247–1293.

[39] M.A. Duran and I.G. Grossmann. "An Outer-approximation Algorithm for a Class of Mixed-Integer Nonlinear Programs". In: *Mathematical Programming* 36.3 (1986), pp. 307–339.

[40] A.C. Scott, J.S. Bennett, and M. Peairs. *The EMYCIN Manual.* Stanford, CA: Department of Computer Science, Stanford University, 1981.

[41] D. Angluin. "Queries and Concept Learning". In: *Machine Learning* 2.4 (1988), pp. 319–342.

[42] *Decision Management for the Masses.* Whitepaper. FICO, 2016.

[43] C.L. Forgy. "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem". In: *Artificial Intelligence* 19.1 (1982), pp. 17–37.

[44] T. Westerlund and K. Lundqvist. *Alpha-ECP, Version 5.101: An Interactive MINLP-Solver Based on the Extended Cutting Plane Method.* Turku, Finland: Åbo Akademi, 2001.

[45] R. Gandy. “Church’s Thesis and the Principles for Mechanisms”. In: *The Kleene Symposium* (1980). Ed. by H.J. Barwise, H.J. Keisler, and K. Kunen, pp. 123–148.

[46] R. L. Sallam and G. Herschel. *Automating Decisions With Intelligent Decision Automation.* Market Analysis. Gartner, 2009.

[47] A. Likas, N. Vlassis, and J.J. Verbeek. “The Global K-means Clustering Algorithm”. In: *Pattern recognition* 36.2 (2003), pp. 451–461.

[48] *The GNU Linear Programming Kit.* Product Description. `http://www.gnu.org/software/glpk/`.

[49] O. Goldreich. *Foundations of Cryptography, Volume 1.* Cambridge, UK: Cambridge University Press, 2001.

[50] O. Goldreich, S. Goldwasser, and S. Micali. “How to Construct Random Functions”. In: *Journal of the ACM* 4.33 (1986), pp. 792–807.

[51] R. Greinier. “The complexity of theory revision”. In: *Artificial Intelligence* 107.2 (1999), pp. 175–217.

[52] *Gurobi Optimizer*. Product Description. `http://www.gurobi.com/products/gurobi-optimizer/`.

[53] B. von Halle. *Business Rules Applied: Building Better Systems Using the Business Rules Approach.* Hoboken, NJ: Wiley, 2001.

[54] B. von Halle and L. Goldberg. *The Business Rule Revolution: Running Business the Right Way.* Cupertino, CA: Happy About, 2006.

[55] D. Harel. "On Folk Theorems". In: *Communications of the ACM* 23.7 (July 1980), pp. 379–389.

[56] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106.

[57] C. Gopal, D. Vesset, and D. Schubmehl. *Worldwide Business Intelligence and Analytics Tools Software Market Shares, 2016: Here Comes the Cloud.* Market Share. IDC, 2017.

[58] S. D. Hendrick and K. E. Hendrick. *The Business Value of Business Rules Management Systems.* Whitepaper. `ftp://ftp.software.ibm.com/software/websphere/dm/The_Business_Value_of_Business_Rule_Management_Systems.pdf`. IDC, 2012.

[59] P. Albert. "ILOG Rules, embedding rules in C++: Results and limits". In: *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)* (1994).

[60] A. Goyal, F. Bonchi, and L.V.S. Lakshamanan. "Learning Influence Probabilities In Social Networks". In: *Proceedings of the third ACM international conference on Web search and data mining (WSMD 2010).* New York, NY: ACM, 2010, pp. 241–250.

[61] *Red Hat JBoss BRMS 6.* DataSheet. `http://www.redhat.com/cms/managed-files/mi-brms-6-datasheet-11436837-inc0357730lw-201603-en.pdf/`. Red Hat, 2016.

[62] E.J. Friedman-Hill. *JESS in Action.* Greenwitch, CT: Manning Publications, 2003.

[63] R.M. Karp. “Reducibility Among Combinatorial Problems”. In: *Complexity of Computer Computations.* New York, NY: Plenum Press, 1972, pp. 85–103.

[64] H.-W. Kelbassa. “Optimal refinement of rule bases”. In: *AI Communications* 17.3 (2004), pp. 123–154.

[65] H.-W. Kelbassa and R. Knauf. “A Process Approach to Rule Base Validation and Refinement”. In: *Proceedings of the Workshop on Knowledge Engineering and Software Engineering at the 28th German Conference on Artificial Intelligence (KI-2005)* (2005), pp. 25–36.

[66] S.C. Kleene. *Introduction to Metamathematics.* Amsterdam, the Netherlands: North-Holland, 1952.

[67] T. Cover and P. Hart. “Nearest Neighbor Pattern Classification”. In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27.

[68] H. Herbst G. Knolmayer. “Business Rules”. In: *Wirtschaftsinformatik* 35.4 (1993), pp. 386–390.

[69] A. Church. “A Set of Postulates for the Foundation of Logic”. In: *Annals of Mathematics* 33.2 (1932), pp. 346–366.

[70] M. Palmirani et al. “LegalRuleML: XML-Based Rules and Norms”. In: *Rule-Based Modeling and Computing on the Semantic Web, Proceedings of the 2011 international conference on Semantic web rules (RuleML 2011)* (2011), pp. 298–312.

[71] A. Johnsen and A.J.R. Berre. “A Bridge between Legislator and Technologist-Formalization in SBVR for Improved Quality and Understanding of Legal Rules”.

In: *Proceedings of the 1st International Workshop on Business Models, Business Rules and Ontologies (BuRO 2010)* (2010).

[72] Y. Lin and L. Schrage. “The global solver in the LINDO API”. In: *Optimization Methods and Software* 24.4-5 (2009), pp. 657–668.

[73] *lp_solve 5.5.2*. Product Description. `http://lpsolve.sourceforge.net/`.

[74] P. Lucas and L. Van Der Gaag. *Principles of Expert Systems.* Boston, MA: Addison-Wesley Longman Publishing Co., 1991.

[75] Y. Matiyasevich. *Hilbert's Tenth Problem.* Boston: MIT Press, 1993.

[76] M. Minsky. *Computation: Finite and Infinite Machines.* London: Prentice-Hall, 1972.

[77] T. Koch et al. “MIPLIB 2010”. In: *Mathematical Programming Computation* 3.2 (2011), pp. 103–163.

[78] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* New York: Springer, 2013.

[79] R. Caruana and A. Niculescu-Mizil. “An Empirical Comparison of Supervised Learning Algorithms”. In: *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)* (2006), pp. 161–168.

[80] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning: An Artificial Intelligence Approach.* San Francisco, CA: Morgan Kaufmann Publishers, 1986.

[81] B. Zadrozny and C. Elkan. “Transforming Classifier Scores into Accurate Multiclass Probability Estimates”.
In: *Proceedings of The 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD 2002).*
Ed. by D. Hand, D. Keim, and R. Ng. New York, NY: ACM, 2002, pp. 694–699.

[82] M. Zur Muehlen and M. Indulska. “Modeling languages for business processes and business rules: A representational analysis”.
In: *Information Systems* 35.4 (2010), pp. 379–390.

[83] E.H. Shortcliffe. *Computer-Based Medical Consultations: MYCIN.*
New York, NY: Elsevier, 1976.

[84] R. Lippmann. “An Introduction to Computing with Neural Nets”.
In: *IEEE Acoustics, Speech and Signal Processing Magazine* 4.2 (1987), pp. 4–22.

[85] A. Newell. “Production Systems: Models of Control Structures”.
In: *Visual Information Processing.* Ed. by W.G. Chase.
New York, NY: Academic Press, 1973, pp. 463–526.

[86] A. Newell and H.A. Simon. *Human Problem Solving.*
Upper Saddle River, NJ: Prentice-Hall, 1972.

[87] M.W. Aiken and O.R. Liu Sheng. “Nexpert Object”.
In: *Expert Systems* 7.1 (1990), pp. 54–57.

[88] A. Niculescu-Mizil and R. Caruana.
“Predicting Good Probabilities with Supervised Learning”.
In: *Proceedings of the 22nd International Conference on Machine Learning (ICML 2005).* Ed. by L. De Raedt and S. Wrobel.
New York, NY: ACM, 2005, pp. 625–632.

[89] J. Nocedal and S. Wright. *Numerical Optimization.* New York, NY: Springer, 2006.

[90] Pierre-André Paumelle. *Proven practices for enhancing performance: A Q&A for IBM WebSphere ILOG JRules 7.0.x.* Redpaper. IBM, 2010.

[91] B. Stineman. *Why WebSphere Operational Decision Management?* White Paper Executive Summary. `ftp://public.dhe.ibm.com/software/emea/de/websphere/WSW14061USEN.pdf/`. IBM Software, 2011.

[92] ONTORULE Project. *ONTOlogies meet Business RULEs.* 2009. URL: `http://ontorule-project.eu/project/description.1.html`.

[93] *OpenRules User Manual.* OpenRules, Inc. Monroe, NJ, 2015.

[94] L. Brownston et al. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1985.

[95] C.L. Forgy. *OPS5 User's Manual.* Pittsburgh, PA: Department of Computer Science, Carnegie-Mellon University, 1981.

[96] *Pega Robotic Process Automation (RPA).* Datasheet. `https://www.pega.com/system/files/docs/2016/Oct/Pega-Robotic-Process-Automation-Data-Sheet.pdf`. Pegasystems, 2016.

[97] J.C. Platt. "Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods". In: *Advances in Large Margin Classifiers.* Ed. by A.J. Smola et al. Cambridge, MA: MIT Press, 1999, pp. 61–74.

[98] G. Plotkin. "A Structural Approach to Operational Semantics". In: *DAIMI FN-19* (1981).

[99] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* New York, NY: Springer-Verlag, 1987.

[100] L. Sterling and E. Shapiro. “Chapter 7: Programming in Pure Prolog”. In: *The Art of Prolog.* Cambridge, MA: The MIT Press, 1999, pp. 129–148.

[101] K. R. Apt and D. Pedreschi. “Reasoning about Termination of Pure Prolog Programs”. In: *Information and Computation* 106.1 (1993), pp. 109–157.

[102] J.R. Quinlan. “Generating Production Rules from Decision Trees”. In: *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI 1987)* 1 (1987), pp. 304–307.

[103] W.N. Venables, D.M. Smith, and the R Core Team. *An Introduction to R.* United Kingdom: Network Theory, 2016.

[104] L. Breiman. “Random Forests”. In: *Machine Learning* 45.1 (2001), pp. 5–32.

[105] L. Raschid. “A Semantics for a Class of Stratified Production System Programs”. In: *Journal of Logic Programming* 21.1 (1994), pp. 31–57.

[106] A.Y. Ng and M.I. Jordan. “On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes”. In: *Advances in Neural Information Processing Systems* 14 (2001), pp. 841–848.

[107] R.G. Ross. *The Business Rule Book.* Houston, TX: Business Rule Solutions LLC, 1994.

[108] R.G. Ross. *Principles of the Business Rule Approach.* Boston, MA: Addison-Wesley Longman Publishing Co., 2003.

[109] J.J. Alferes et al., eds.
Rule Technologies. Research, Tools, and Applications. 10th International RuleML Symposium (RuleML 2016). Berlin, Germany: Springer, 2016.

[110] P. Compton, Y.S. Kim, and B.H. Kang.
"Linked Production Rules: Controlling Inference with Knowledge". In: *Knowledge Management and Acquisition for Smart Systems and Services.* Ed. by Y.S. Kim, B.H. Kang, and D. Richards.
13th Pacific Rim Knowledge Acquisition Workshop (PKAW 2014). New York, NY: Springer, 2014, pp. 84–98.

[111] T. Achterberg. "SCIP: Solving Constraint Integer Programs".
In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41.

[112] *ServiceNow Platform Business Rules.* Product Documentation.
`https://docs.servicenow.com/bundle/helsinki-servicenow-platform/page/script/business-rules/concept/c_BusinessRules.html/`.
ServiceNow.

[113] C. Shannon. "A Universal Turing Machine with Two Internal States".
In: *Annals of Mathematics Studies* 34 (Automata Studies) (1956). Ed. by McCarthy J. Shannon C., pp. 157–165.

[114] S. Amaran et al.
"Simulation optimization: a review of algorithms and applications".
In: *4OR* 12.4 (2014), pp. 301–333.

[115] J.C. Nash. "The (Dantzig) Simplex Method for Linear Programming".
In: *Computing in Science and Engineering* 2.1 (2000), pp. 29–31.

[116] J. Sneyers, T. Schrijvers, and B. Demoen. "The Computational Power and Complexity of Constraint Handling Rules". In: *Proceedings of the 2nd Workshop on Constraint Handling Rules* (2005), pp. 3–17.

[117] *IBM SPSS Modeler and SPSS Analytic Server: Big data simplified.* Data Sheet. `https://public.dhe.ibm.com/common/ssi/ecm/yt/en/ytd03323usen/YTD03323USEN.PDF/`. IBM, 2016.

[118] *SupportPac LB02 Version 2.1: WebSphere Operational Decision Management Integration with the SPSS Predictive Analytics Suite.* Product Documentation. `ftp://public.dhe.ibm.com/software/integration/support/supportpacs/individual/lb02.pdf/`. IBM, 2012.

[119] G. Lausen, B. Ludäscher, and W. May. "On Active Deductive Databases: The Statelog Approach". In: *Transactions and Change in Logic Databases (DYNAMICS).* Ed. by B. Freitag et al. Berlin, Germany: Springer, 1998, pp. 69–106.

[120] M.A. Hearst et al. "Support Vector Machines". In: *IEEE Intelligent Systems and their Applications* 13.4 (1998), pp. 18–28.

[121] J. Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96.

[122] I. Horrocks et al. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* W3C Member Submission. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`. W3C, 2004.

[123] *TIBCO ActiveMatrix BPM Concepts.* Product Documentation. TIBCO, 2017.

[124] J.R. Quinlan. "Simplifying decision trees". In: *International Journal of Man-Machine Studies* 27.3 (1987), pp. 221–234.

[125] M. Triska. "Correctness Considerations in CLP(FD) Systems". PhD thesis. Vienna University of Technology, 2014.

[126] A. Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42.1 (1937), pp. 230–265.

[127] A. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42 (1937), pp. 230–265.

[128] A. Turing. “Systems of Logic Based on Ordinals”. PhD thesis. Princeton University, 1939.

[129] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 1979.

[130] L. Liberti and F. Marinelli. “Mathematical programming: Turing completeness and applications to software analysis”. In: *Journal of Combinatorial Optimization* 28.1 (2014), pp. 82–104.

[131] T. Hastie, R. Tibshirani, and J. Friedman. “Unsupervised Learning”. In: *The Elements of Statistical Learning.* New York, NY: Springer, 2009. Chap. 14, pp. 485–585.

[132] L.G. Valiant. “A Theory of the Learnable”. In: *Communications of the ACM* 11.27 (1984), pp. 1134–1142.

[133] A. Van Gelder. “Efficient Loop Detection in Prolog using the Tortoise-and-hare technique”. In: *Journal of Logic Programming* 4.1 (1987), pp. 23–31.

[134] V.N. Vapnik. *The Nature of Statistical Learning Theory.* New York, NY: Springer-Verlag, 1995.

[135] S. Verbaeten, K. Sagonas, and D. De Schreye. “Modular termination proofs for Prolog with tabling”. In: *Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP’99).* Ed. by G. Nadathur. Berlin, Germany: Springer-Verlag, 1999, pp. 342–359.

[136] V. Vianu. “Rule-Based Languages”. In: *Annals of Mathematics and Artificial Intelligence* 19.1-2 (1997), pp. 215–259.

[137] A. Paschke, G. Hallmark, and C. De Sainte Marie. *RIF Production Rule Dialect (Second Edition).* W3C Recommendation. `http://www.w3.org/TR/2013/REC-rif-prd-20130205/`. W3C, 2013.

[138] O. Wang et al. “Controlling the Average Behavior of Business Rules Programs”. In: *Rule Technologies. Research, Tools, and Applications.* Ed. by J.J. Alferes et al. 10th International RuleML Symposium (RuleML 2016). Berlin, Germany: Springer, 2016, pp. 83–96.

[139] O. Wang and L. Liberti. “Controlling Some Statistical Properties of Business Rules Programs”. In: *Proceedings of the 11th International Conference on Learning and Intelligent Optimization (LION 11).* New York, NY: Springer, 2017.

[140] O. Wang et al. “The Learnability of Business Rules”. In: *Second International Workshop on Machine Learning, Optimization, and Big Data (MOD 2016).* New York, NY: Springer, 2016, pp. 257–268.

[141] D.A. Waterman. “Adaptive production systems”. In: *Proceedings of the 4th international joint conference on Artificial intelligence (IJCAI 1975) - Volume 1.* San Francisco, CA: Morgan Kaufmann Publishers, 1975, pp. 296–303.

[142] H.P. Williams. *Model Building in Mathematical Programming.* 4th. Chichester: Wiley, 1999.

[143] J. McDermott. "R1: An Expert in the Computer Systems Domain". In: *Proceedings of the 1st National Conference on Artificial Intelligence (AAAI 1980)*. Menlo Park, CA: AAAI Press, 1980.

Titre : Modèles de Règles Adaptatifs : Apprentissage Statistique pour les Systèmes à Base de Règles
Mots clés : règles métiers, apprentissage automatique, apprentissage à but statistique, optimisation

Résumé : Les Règles Métiers (Business Rules en anglais, ou BRs) sont un outil communément utilisé dans l'industrie pour automatiser des prises de décisions répétitives. Le problème de l'adaptation de bases de règles existantes à un environnement en constante évolution est celui qui motive cette thèse. Des techniques existantes d'Apprentissage Automatique Supervisé peuvent être utilisées lorsque cette adaptation se fait en toute connaissance de la décision correcte à prendre en toute circonstance. En revanche, il n'existe actuellement aucun algorithme, qu'il soit théorique ou pratique, qui puisse résoudre ce problème lorsque l'information connue est de nature statistique, comme c'est le cas pour une banque qui souhaite contrôler la proportion de demandes de prêt que son service de décision automatique fait passer à des experts humains. Nous étudions spécifiquement le problème d'apprentissage qui a pour objectif d'ajuster les BRs de façon à ce que les décisions prises aient une valeur moyenne donnée.
Pour ce faire, nous considérons les bases de Règles Métiers en tant que programmes. Après avoir formalisé quelques définitions et notations dans le Chapitre 2, le langage de programmation BR ainsi défini est étudié dans le Chapitre 4, qui prouve qu'il n'existe pas d'algorithme pour apprendre des Règles Métiers avec un objectif statistique dans le cas général. Nous limitons ensuite le champ d'étude à deux cas communs où les BRs sont limités d'une certaine façon : le cas Borné en Itérations dans lequel, quelles que soit les données d'entrée, le nombre de règles exécutées en prenant la décision est inférieur à une borne donnée ; et le cas Linéaire Borné en Itérations dans lequel les règles sont de plus écrite sous forme Linéaire. Dans ces deux cas, nous produisons par la suite un algorithme d'apprentissage basé sur la Programmation Mathématique qui peut résoudre ce problème. Nous étendons brièvement cette formalisation et cet algorithme à d'autres problèmes d'apprentissage à objectif statistique dans le Chapitre 5, avant de présenter les résultats expérimentaux de cette thèse dans le Chapitre 6.

Title: Adaptive Rules Models: Analytics Learning for Rule-Based Systems
Keywords: business rules, machine learning, statistical goal learning, optimization

Abstract: Business Rules (BRs) are a commonly used tool in industry for the automation of repetitive decisions. The emerging problem of adapting existing sets of BRs to an ever-changing environment is the motivation for this thesis. Existing Supervised Machine Learning techniques can be used when the adaptation is done knowing in detail which is the correct decision for each circumstance. However, there is currently no algorithm, theoretical or practical, which can solve this problem when the known information is statistical in nature, as is the case for a bank wishing to control the proportion of loan requests its automated decision service forwards to human experts. We study the specific learning problem where the aim is to adjust the BRs so that the decisions are close to a given average value.
To do so, we consider sets of Business Rules as programs. After formalizing some definitions and notations in Chapter 2, the BR programming language defined this way is studied in Chapter 3, which proves that there exists no algorithm to learn Business Rules with a statistical goal in the general case. We then restrain the scope to two common cases where BRs are limited in some way: the Iteration Bounded case in which no matter the input, the number of rules executed when taking the decision is less than a given bound; and the Linear Iteration Bounded case in which rules are also all written in Linear form. In those two cases, we later produce a learning algorithm based on Mathematical Programming which can solve this problem. We briefly extend this theory and algorithm to other statistical goal learning problems in Chapter 5, before presenting the experimental results of this thesis in Chapter 6. The last includes a proof of concept to automate the main part of the learning algorithm which does not consist in solving a Mathematical Programming problem, as well as some experimental evidence of the computational complexity of the algorithm.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France