ALMA MATER STUDIORUM—UNIVERISITÀ DI BOLOGNA

Dottorato di Ricerca in Data Science and Computation
XXXIII Ciclo

On the role of Computational Logic in Data Science: representing, learning, reasoning, and explaining knowledge

Settore Concorsuale:
09/H1 – Sistemi di Elaborazione delle Informazioni

Settore Scientifico Disciplinare:
ING-INF/05 – Sistemi di Elaborazione delle Informazioni

Coordinatore Dottorato
Prof. Andrea Cavalli

Candidato
Giovanni Ciatto

Supervisore
Prof. Andrea Omicini

Esame Finale Anno 2022

Abstract

In this thesis we discuss in what ways computational logic (CL) and data science (DS) can *jointly* contribute to the management of knowledge within the scope of modern and future artificial intelligence (AI), and how technically-sound software technologies can be realised along the path. An agent-oriented mindset permeates the whole discussion, by stressing pivotal role of autonomous agents in exploiting both means to reach higher degrees of intelligence. Accordingly, the goals of this thesis are manifold. First, we elicit the analogies and differences among CL and DS, hence looking for possible synergies and complementarities along 4 major knowledge-related dimensions, namely representation, acquisition (a.k.a. learning), inference (a.k.a. reasoning), and explanation. In this regard, we propose a conceptual framework through which bridges these disciplines can be described and designed. We then survey the current state of the art of AI technologies, w.r.t. their capability to support bridging CL and DS in practice. After detecting lacks and opportunities, we propose the notion of *logic ecosystem* as the new conceptual, architectural, and technological solution supporting the *incremental* integration of symbolic and sub-symbolic AI. Finally, we discuss how our notion of logic ecosystem can be reified into actual software technology and extended towards many DS-related directions.

Keywords: Computational Logic · Data Science · XAI · Logic Ecosystem · 2P-Kt

Odi et amo. Quare id faciam fortasse requiris.
Nescio, sed fieri sentio et excrucior.

Acknowledgements

This thesis and my whole PhD wouldn't have been the same without the many enlightening people I've met along the path. Hence, this section is my way of expressing my gratitude and esteem for the many mentors, colleagues, and friends I had the pleasure to work with in the last five years.

Firstly, I'd wish to thank my supervisor and mentor, prof. Andrea Omicini, for the many years under his guide, and the many enlightening pieces of wisdom, and suggestions about the research, the academic career, and everyday life—other than, of course, the infinitely many ponsense nuns[1]. He showed to me how to become a fierce and autonomous researcher, as well as the importance of conceiving science as a community effort, and he gave me a lot of opportunities along the way. Of all such things, I'm sincerely grateful. Should I start another PhD (so to speak), I would choose his supervision once again.

Concerning mentorship, I honestly owe my gratitude to prof. Mirko Viroli as well, for trusting me and letting me join the academic world back in 2017. Revenge shall be sought, but up to then I'd wish to thank him for the many opportunities he gave me along the years, and for the many discussions and lunches we have shared. For similar reasons, and for being a continuous source of inspiration, I'd wish to thank professors Alessandro Ricci, and Enrico Denti.

Concerning inspiration, as special mention is due to a few post-doc colleagues who have had quite an impact in shaping what my idea of "a good researcher" actually is. Accordingly, I'd with to thank dr. Danilo Pianini for being such an example of dedication to scientific and technical knowledge, dr. Stefano Mariani for helping me with my first steps in research and dr. Roberta Calegari for always supporting me since then. Finally, I'd wish to thank dr. Davide Calvaresi, for being such a great friend and co-author, as well as dr. Sara Montagna for all the precious discussions we shared.

There are also a number of friends I've spent some wonderful time with in the many laboratories I've worked in, along the years. These include, in casual order, dr. Matteo Francia, dr. Lorenzo Monti, dr. Roberto Casadei, dr. Angelo Croatti, Niccolò Marini. Them I'd wish to thank for all the wonderful time spent talking

[1]nonsense puns

about life, politics, and the glory and misery of a PhD life—other than, from time to time, research.

Furthermore, I'd wish to thank all the brilliant students I had the pleasure to supervise for their theses or graduate projects, to whom I have taught at least as much as I have learn from them. These include, but are not limited to, Alfredo Maffi, Lorenzo Rizzato, Luca Tremamunno, Jason Dellaluce, Federico Siboni, Giuseppe Pisano, Federico Sabbatini, Andrea Agiollo, Giovanni Speciale, Matteo Castigliò, Andrea Giordano, and Matteo Magnini.

Finally, after a number of theses dedicated to my parents and sister as the family I've been risen by – which I still thank, of course –, I'd wish to dedicate this thesis and my whole PhD to the family I've created along the path, i.e. my beloved partner, Elena Lucarella, who shared with me all the joy and sorrow, as well all the successes and failures I've encountered along the way. I'd wish to thank her the most, for supporting me since the very beginning, for always pushing me to go the extra mile, for being there every time I felt lost, and for being such a great confidant and friend—other than lover. I wouldn't have made it this far without her, nor I'd wish.

Giovanni Ciatto, April 17, 2022

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In the last decade, we have witnessed an explosion in the exploitation of artificial intelligence (AI) both in the academy and in the industry, and in virtually all strategical sectors of human expertise. This is not the first time in history that AI attains unprecedented levels of attention, expectations, and funding, yet it is the first time that such momentum is driven by a pervasive adoption of data science (DS) and, in particular, machine learning (ML).

Nowadays, the tree terms – AI, DS, and ML – are often used mistakenly interchangeably, especially by practitioners. Should we speculate on what the causes of such phenomenon are, we would argue this is likely due to the strong hype characterising modern data-driven solutions—both in theory and in practice. This leads both researchers and practitioners to focus on the development of *ML-oriented* frameworks or technologies which, in turn, create a sampling bias making people think that ML exhausts DS, and DS saturates AI. As we further discuss in the subsequent chapters, this is really far from the truth. There are many interesting aspects of AI which lay outside the realm of DS. Notably, in this thesis we focus on computational logics (CL) – a prominent aspect of AI populating the portion which is not covered by DS – and its potential role in complementing DS.

As sub-fields of AI, both DS and CL share the common goal of mimicking human intelligence. Of course, they do so in different ways. They focus on different notions and aspects of intelligence, they pursue intelligence through different ways, and for different purposes. Notably, most differences lay in the way CL and DS treat *knowledge*, and, in particular, in the way knowledge is represented, acquired, manipulated, and transferred.

CL, for instance, focuses on *rational* intelligence, and it aims at endowing machines with human-like, automated *reasoning* capabilities. Following this purpose, it relies on *symbolically* represented knowledge, either acquired via logic induction or via manual handcrafting, manipulated via logic inference (e.g. deduction or abduction), and transferred by simply presenting symbols into shared formats.

Dually, DS focuses on *intuitive* intelligence, and it aims at endowing humans with statistical tools for mining significant and predictive information from data, in a principled way. When applied to machines, DS provides them with powerful pattern matching, recognition, or stimulus-response capabilities. For this reason, it relies on sub-symbolically (e.g. *numerically*) represented knowledge, commonly acquired from data via ML, manipulated via algebraic or differential operations, and transferred in disparate, purpose-specific ways.

Of course, both CL and DS come with shortcomings. On the one side, CL commonly requires *(i)* some symbolic knowledge to be eventually handcrafted by humans, manually; and *(ii)* the task at hand to have a clear formulation, which can be expressed via crisp symbols. The former issue, clearly hinders scalability, making CL fall short on the knowledge provisioning side. Vice versa, DS is very well suited on this side, as it naturally leverages on scalable algorithms which are designed to mine information semi-automatically from data, possibly scaling up to very large datasets. The latter issue, in turn, makes CL poorly suited to handle fuzzy tasks which are hard to formalise or encode symbolically—think, for instance, to the task of handwritten digits recognition. On the other side, to be effective, DS commonly requires *(i)* very large amounts of data; and *(ii)* users to be willing and capable of interpreting the numeric results it outputs. The former issue actually constrains the exploitation of DS into use cases where data is already available or a provisioning procedure is admissible. Vice versa, CL is data efficient as it can bring valuable results even in presence of very small prior knowledge. In turn, the interpretability issue is nowadays among the most relevant topics. Given the wide exploitation of DS in so many areas of expertise, clarity and intelligibility of its outcomes are becoming a critical aspects—mostly because of their sub-symbolic nature. Conversely, CL is inherently symbolic in nature and therefore less subject to such interpretability issues.

Accordingly, this thesis stems from the acknowledgement that CS and DS are complementary – rather than competing – aspects of AI, and that *knowledge* plays a pivotal role in both these fields. Along this line, we aim to *elicit* and *enable* the many possible bridges among them, w.r.t. knowledge manipulation. In doing so, we follow the ultimate purpose of increasing the degree of intelligence and autonomy of modern computational systems. Therefore, our focus is on computational entities and on the ways they can combine and integrate CL and DS to either act more intelligently or more autonomously—or both.

On the one side, we *elicit* analogies, dichotomies, and possible synergies among CL and DS by analysing them along four orthogonal dimensions, corresponding to as many knowledge-related activities, namely:

representation — i.e. the way knowledge is expressed and made interpretable by either machines or human beings, or both; e.g. via symbols, formulæ, or tensors of real numbers

acquisition (a.k.a. learning) — i.e. the way novel knowledge is learned from prior information, mined from data, or attained from external sources; e.g. via data mining, via induction, or via interaction

inference (a.k.a. reasoning) — i.e. the way decisions, suggestions, recommendations, or predictions can be *automatically* computed out of prior knowledge; e.g. via automated deduction/abduction, or via classification/regression

explanation — i.e. the way knowledge can be transferred to another entity – be it computational or human –, in such a way that the recipient can take advantage of it, similarly to how the provider would

On the other side, we acknowledge that both CL and DS have a prominent overlap with computer science (CS) and software engineering (SE). Regardless of how they manipulate knowledge, both approaches subtend a mathematical modelling of many computational aspects, which must then be reified into well-engineered software technologies to let practitioners actually exploit them. Along this line, we further analyse CL and DS from both a *computational* and *technological* perspective. While the computational perspective focuses on *what* data structures, algorithms, and workflows they leverage upon to attain intelligence, the technological perspective focuses on *how* such aspects can be translated in practice, via robust software architectures and effective implementations. Along this line, in particular, we assess the current state of the art for technologies laying at the intersection among DS and CL – or supporting the construction of bridges among the two fields –, identifying holes and proposing lacks to overcome them. The latter in particular is the contribution by which we *enable* the actual combination of CL and DS in practice.

We carry out the whole discussion under an agent-oriented mindset. Within the scope of this document, we call "agent" any autonomous entity having its own *locus of control*—be it a human being or a running process programmed software. We may refer to agents as "intelligent" in case the come equipped with human-like knowledge-related capabilities, such as the aforementioned capabilities of representing, learning, inferring, or explaining knowledge—or, possibly, a multitude of them. Under such a mindset, human agents are intelligent by definition, whereas software agents may tend to intelligence by acquiring one or more of these capabilities via either CS or DS—or, hopefully, a combination of them. Hence, what we have so far called "machines" are indeed "software agents", and the overall role of the agent-oriented mindset is about focussing on *who* is charge of manipulating knowledge and *when*.

Goals of the thesis. Summarising, the whole thesis discusses in what ways CL and DS can jointly contribute to the management of knowledge within the scope of modern and future intelligent systems, and how technically-sound software technologies can be realised along the path. An agent-oriented mindset permeates the whole discussion, by stressing pivotal role of autonomous agents in exploiting both means to reach higher degrees of intelligence.

Along this line, the goals of this thesis can be concisely enumerated as follows:

1. understanding the analogies and differences among CL and DS, hence looking for possible synergies and complementarities;

2. derive a conceptual framework through which bridges among CL and DS can be described and designed, along the 4 main dimensions;

3. evaluate the current state of the art of AI technologies, w.r.t. their capability to support the realisation of those bridges;

4. propose the notion of *logic ecosystem* as the new conceptual, architectural, and technological solution supporting the *incremental* integration of symbolic and sub-symbolic AI;

5. discuss how our notion of logic ecosystem can be *(i)* reified into actual software technology and *(ii)* extended to support the joint exploitation of DS and CL in practice.

Notably, goals 1 and 2 deal with the computational perspective, whereas goals 3–5 deal with the technological one.

Structure of the thesis. Accordingly, this thesis is organised in two parts, namely "What" and "How".

What. In part I, we focus on the computational perspective. This is where we address goals 1 and 2.

Our discussion starts from chapter 2, where we provide a brief recap of the history of AI, with a focus on the symbolic–connectionist dichotomy. There, we introduce the two main branches of AI—namely, the *symbolic* and *sub-symbolic* (initially referred to as "connectionist") ones.

In the subsequent chapters, we recall classical definitions and survey the current *theoretical* state of the art for both CL and DS. In doing so, we present and compare their strengths and weaknesses, and the many possible bridges among them, w.r.t. the 4 principal dimensions—namely, knowledge representation (chapter 3), learning (chapter 4), inference (chapter 5), and explanation (chapter 6).

How. Conversely, in part II, we focus on the technological perspective. This is where we address goals 3, 4, and 5.

Our discussion starts from chapter 7, where we discuss the role of logic-based approaches and technologies in the modern AI playground. Then, to fully accomplish goal 3, we provide a state of the art for logic-based technologies in chapter 8. There, we focus on logic-based technologies coming some actually usable reification in software, and identify the current lacks w.r.t. recent theoretical advances.

In chapter 9, we introduce our notion of logic ecosystem, and we propose 2P-Kt as its main reification in software, hence addressing goal 4. Then, in chapters 10–15, we present a number of possible ways to extend the 2P-Kt ecosystem towards DS. A detailed overview of these chapters is provided in section 9.4. Notably, chapter 15 is entirely dedicated to future research directions.

Finally, chapter 16 concludes the thesis and summarises our contribution.

Part I

What

Chapter 2

Historical Perspective on AI

AI is a multi-faceted discipline leveraging on contributions coming from several areas of human knowledge, there including Mathematics, Computer Science, Statistics, Psychology, Philosophy, and many others. A famous and comprehensive survey on the many aspects of AI is proposed by Russell and Norving in [RN16]. In the second half the 20th century – when AI was firstly recognised as discipline by itself – several approaches towards machine intelligence became subject of intensive research efforts, leading to the vast corpus of literature and to the abundance of techniques available today.

Notably, two main families of approaches has initially emerged in AI, namely, the *symbolic* and *connectionist* ones [Smo87, Sun01]. While the former focuses on representing the world through symbols – in turn representing concepts –, thus emulating how the human *mind* reasons and infers, the latter aims at mimicking human intuition by emulating how the human *brain* works at a very low level. Despite both families have both pros and cons, they have stepped through both glory and misery—in terms of expectations, funding, research interest, and industry adoption [Hen08, RN16].

For instance, despite the initial hype, artificial neural networks (NN) – the warhorse of connectionism – encountered their first *winter* when Rosenblatt's *perceptron* [Ros57] was proven unable to learn the XOR function [MP88]. The period following the publication of the well-known back-propagation algorithm [RHW86] and the proof that multi-layered perceptrons could be used as universal functional approximators [Cyb89], can be considered as the second *spring* of connectionist approaches. However, at the time – likely, because of computational limits of the hardware and the lack of data – the success of connectionist approaches can be considered very moderate, especially when compared with the explosion of deep NN and deep learning (DL) [GBC16] which was pervasive in both the academy and the industry during the 2010s, and can thus be considered the third spring of AI.

Even if it is currently not as popular as NN, the history of symbolic AI is extremely important as well—mostly because of the prominent influence it has on the many fields converging in AI. Despite their original ambition of reproducing human reasoning *in toto* has been inevitably rejected by facts, symbolic approaches gave birth to several research lines which are nowadays considered autonomous fields, such as computational logic [Llo90], logic programming [Apt90], planning [RN16, Chap. 10-11], multi-agent systems [FW99], etc.

A few decades later, many things has changed. ML, DL, Data Mining [Han06], and Bayesian Inference have enormously widened the spectrum of tasks AI can handle, other than the amount of use cases where AI can be applied. Nevertheless, a dualism is still there, alive and healthy, dividing symbolic approaches from what are now called *sub-symbolic* ones.

Nowadays, sub-symbolic techniques include NN, but they are not limited to the connectionist techniques. The panorama of sub-symbolic techniques has been widened by the development of several data-mining algorithms – along with their efficient implementations –, such as SVM [SS04], K-Means, Expectation Maximisation [DLR77], Viterbi [Vit06], etc., which mostly leverage on *numerical* computations while not being backed by a biological metaphor.

Summarising, at the gates of 2020s, AI consists of a number of powerful techniques – often backed by sound theoretical or empirical backgrounds –, which are widely employed to automate disparate tasks, both in the industry and in the research. Such tasks, and, in particular, the techniques supporting them, can be categorised within two mostly disjoint families—namely the symbolic and sub-symbolic ones.

Weak vs. Strong AI

A fundamental question in AI concerns the ultimate goal of the discipline itself. Some say AI should (tend to) produce machines which are actually able to think intelligently – thus adapt to different situations, understand the context their are situated into, learn from the interactions with the environment and with others –, while say it would be sufficient to create machines simply acting *as if* they were thinking intelligently. The former perspective is classically known as *strong* AI, while the latter is known as *weak* AI. Searle's Chinese Room argument [Sea80] clearly explains the difference among the two by means of a practical example, whereas the well known Turing test [Tur50] provides a practical means to decide whether a machine's AI is actually strong or not.

Even without discussing the many important and subtle philosophical issues arising from such a dualistic view of AI, we note what follows. The original goal of AI was reaching strong AI. This is likely why, initially, so much hype was put in both symbolic and connectionist approaches. But this is also justifying the strong disappointment which led AI towards its first winter. Most researchers soon realised that weak AI was a far more affordable deal, and this is likely why they stopped seeking generality and started focusing on how to improve each single technique, tailoring them to the domains where they could bring more advantage.

A few years later, the global effect is that a plethora of techniques is available to effectively and efficiently tackle as many tasks. But the glue keeping everything together is still human intelligence [YWC+18]. Indeed, symbolic techniques still require human beings to *handcraft* most complex rules or to *manually* build large knowledge bases. Similarly, most ML-powered models still rely on data scientists to lead their training process. Data scientists are still needed, for instance, to clean-up and pre-process data, and to set up predictors hyper-parameters through their experience, or to leverage on their intuition to interpret how a trained predictor is functioning.

Summarising, the success of AI nowadays is also due to its reduced expectations with respect to what can be delegated to machines. As a side effect, poor care is dedicated in studying how AI could be used to *automate* the many processes involving several, interrelated AI-powered tasks.

Symbolic vs. Sub-symbolic AI

In the recent years, the historical dichotomy between the "two souls" of AI has been reconciled, in favour of a comprehensive vision where symbolic and sub-symbolic approaches are seen as *complementary* – rather than in a competition – so that they mutually soften their corners [HQR17, CCDO19, CCM+18]. While symbolic approaches are well suited for relatively small-sized problems where complex and exact tasks has to be performed, possibly relying on structured data – like for instance planning a sequence of actions, finding a path in a graph taking several constrains into an account, deducing information from a prior knowledge base, or learning mathematical relations from vary small data sets –, sub-symbolic approaches are best suited for those use cases where big (up to huge) amounts of possibly unstructured data must be processed, where errors or lack of precision is tolerated to some extent, if unavoidable—like for instance classifying images or texts, profiling customers by looking at their shopping history, forecasting the weather for a particular area, etc. Such issues, are not affecting symbolic techniques—especially when symbols are wisely chosen in order to steer humans' intuition. This is because symbols are far closer to what our conscious, rational mind used to handle. For all such reasons, many researchers along the history of AI have argued that a comprehensive approach unifying the two worlds would bring great advantage.

More precisely, complementarity between symbolic and sub-symbolic AI naturally emerges when comparing the two approaches under the following perspectives:

- sub-symbolic AI is *opaque*, meaning that human beings struggle in understanding the functioning and behaviour of sub-symbolically intelligent systems; instead, symbolic AI is more *transparent*, as it is both human- and machine-interpretable at the same time

- sub-symbolic AI can improve itself *automatically* by consuming data, but it is difficult to extend and re-use outside the contexts it was designed for; conversely, symbolic AI is flexible and extensible, but requires humans to *manually* provide symbolic knowledge

- sub-symbolic AI is adequate for *fuzzy* problems where some (minimal) degree of error or uncertainty can be tolerated; whereas symbolic AI calls for precise data and queries provided by human beings, yet provides exact, *crisp* results as its outcome.

In the remainder of this chapter, we provide a brief overview of the two major disciplines laying within the scope of symbolic or sub-symbolic AI, respectively. These are computational logic – a prominent branch of symbolic AI leveraging on logics to per form any knowledge-related task, ranging from representation to inference –, and data science – a prominent branch of sub-symbolic AI leveraging on statistics to manipulate data and mine knowledge out of them.

2.1 Computational Logic

This section contains contributions from the following works of ours: [CCO21a, KBB$^+$21, CCDO20]

Computational logic (CL) [Llo90] is a fundamental research area for *artificial intelligence* (AI), dealing with formal logic as a means for computing [Pau18]. Its penetration into *symbolic AI* is nearly pervasive nowadays, and increasingly going deeper within *sub-symbolic AI* [CCO20, CCMO21a]: CL has enabled the development of the former in the past, and it is now pushing the latter towards interpretability and explainability. Be it exploited to manipulate symbols, or to make sub-symbolic solutions human-intelligible, the common expectation behind CL is to endow software systems with *automated* reasoning.

Generally speaking, automated reasoning involves three major aspects: *(i)* logics, *(ii)* inference rules, and *(iii)* resolution strategies.

Logic formally defines how knowledge is represented and how novel knowledge can be derived from prior one. Each logic comes with several *inference rules*, dictating how to produce new knowledge under particular circumstances. When coupled with some *resolution strategy*, inference rules become deterministic algorithms that computers can execute to reason autonomously.

Many logics exist in CL – e.g. propositional, first-order (FOL), temporal, deontic, etc. –, each one targeting a specific way of reasoning. For instance, temporal logic enables reasoning about the chronological ordering of events, deontic logic supports reasoning about permissions/prohibitions and their circumstances, while FOL is general-purpose. Furthermore, different sorts of inference rules exist for different logics. Some are *deductive* – drawing conclusions out of premises –, some are *inductive* – looking for general rules out of several premises-conclusion examples –, while other are *abductive*—speculating on which premises caused some conclusions. Finally, when a resolution strategy exists for some rule and logic, it can be reified in software, and used to build intelligent systems capable of automated reasoning. Software of that sort are commonly referred to as a part of the *logic programming* (LP) paradigm [MN96].

In LP, programs are typically *theories* (a.k.a. *knowledge bases*, KB), i.e. collections of sentences in logical form, expressing *facts* and *rules* about some domain, typically in the form of *clauses*, i.e. expressions connecting a number of interrelated *predicates* via logic connectives (e.g. operators such as $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, etc.). There, predicates represent statements describing or relating one or more entities about the domain at hand.

Depending on the particular logic of choice, predicates – and therefore clauses – may, carry *variables*, i.e. placeholders for unknown entities, and possibly quantifiers for those variables ($\exists$ or $\forall$). Some logics may also endow clauses with further information, such as for instance *probabilities* – describing the degree of likelihood for a clause to hold true –, or *modalities*—describing the context in which a clause may hold (e.g. *when*). In any case, logic information is represented in such a way that both human and computational agents can interpret and manipulate it, in principle.

One powerful trait of logics is that they enable the representation of complex, intricate, or infinite domains *intensively* (i.e. implicitly) rather than explicitly—e.g. via multiple recursive clauses. So, if a domain involves an infinite amount of entities, these do not necessarily require an infinite amount of memory to be represented. For instance, the set of natural integers can be represented in logic using just two FOL clauses—of which, one is recursive.

Software agents devoted to automated reasoning via LP are commonly referred to as logic *solvers*. They rely on pre-existing KB to derive inferences via some inference procedure and resolution strategy. They may do so either *reactively*, – i.e. in response to some external stimulus, e.g. some user's *query* –, or *pro-actively*—i.e. spontaneously, in order to reach some goal, e.g. computing the optimal path before moving. Prolog-based solvers [CR93, Col86], for instance, exploit a *deductive* procedure rooted into the SLDNF resolution principle [Kow74, Cla77], and a depth-first strategy. They commonly do so in response to users' queries, provided via a textual interface. Yet, a number of Prolog solvers exists supporting the same inference procedure via different strategies (e.g. tabled resolution [CW96, SW12]) or as well as entirely different principles (e.g. Constraint Logic Programming). Of course, other options exist targetting other logics as well, like, e.g., abductive [FK97], inductive [Md94], probabilistic [dK15] inference. Each of them represents a particular reification of a logic solver.

Limits of CL. Despite the many possibilities, however, there are a number of issues which are not tied to any particular choice of logic, inference procedure, or resolution strategy, but they are rather inherent to CL itself. Such issues involve *(i)* decidability, *(ii)* tractability, *(iii)* knowledge acquisition, and *(iv)* symbols grounding.

Decidability and tractability deal with the theoretical questions: "can a logic solver provide an answer to any logic query it receives? can it do so in reasonable time?". Such aspects are deeply entangled with the particular logic the solver is leveraging upon. Depending on which and how many features a logic includes, it may be more or less *expressive.* The higher the expressiveness, the more the complexity of the problems which may be represented via logic and processed via inference increases. This opens to the possibility, for the solver, to meet queries which cannot be answered in useful time, or relying upon a limited amount of memory, or at all. Roughly speaking, more expressive logic languages make it easier for human beings to describe a particular domain – usually, requiring them to write less and more concise clauses –, at the expense of a higher difficulty for software agents to draw inferences autonomously—because of computational *tractability.* This is a well-understood phenomenon in both CS and CL [LB87, BL04], often referred to as the *expressiveness–tractability* trade-off. In practice, however, a good trade-off is represented by FOL and its subsets (e.g. Horn logic [Mak87]), or modal variants (e.g. linear temporal logic [Pnu77]). Despite consisting of Turing-equivalent formalisms – for which the existence of undecidable or intractable situations cannot be excluded in the general case –, they come with sufficiently wide representational capabilities and effective inference procedures, making them exploitable in practice—provided that human developers avoid writing undecidable/intractable algorithms.

Knowledge acquisition deals with the question "where does the knowledge solvers reason upon come from", or alternatively: "who is in charge of constructing knowledge bases"? Recalling that logic clauses may become arbitrarily complex and represent possibly infinite domains in a very concise way, it is unsurprising that the burden of knowledge production is mostly on humans. Unfortunately, this implies the degree of automatism in knowledge production is pretty low, as well as the scalability of the approach. Many attempts have been performed over the years to distil human knowledge into symbolic form to formalise common-sense for software agents. To date, there exist a number of common-sense knowledge bases and ontologies, supporting practical textual-reasoning tasks on real-world documents including analogy-making, and other context oriented inferences—see for instance [LLSB04, TL18, LS04, LLS02, Sha00]. Yet, most of these solutions are either semi-automatically constructed, or community driven – when not both –, therefore exhaustiveness, consistency, or coherence may be lacking. There have also been a number of attempts to construct very large knowledge bases of common-sense knowledge by hand, one of the largest being the CYC program by Douglas Lenat at CyCorp [Len95]—which is, however, only usable behind payment.

Finally, symbols grounding deals with the problem of letting software agents provide semantics for the symbols they manipulate. Put it simply, we may tell a logic solver that Abraham is the father of Isaac – $father(\texttt{isaac}, \texttt{abraham})$ –, and also that, for all possible X and Y, if X is the father of Y, then Y must be the child of X – $father(X, Y) \rightarrow child(Y, X)$ –, and the solver may also be able to infer that Isaac is thus the child of Abraham, while still having no idea of how to recognise Isaac, Abraham, nor the fatherhood relation, were it written in another form. In other words, the bindings between the symbols processed by software agents and the corresponding entities from the real world are hard to establish and maintain for a bare logic solver—unless other mechanisms are in place. This issue will hardly be solved within the symbolic world alone, as it is deeply entangled with the problem of letting a software agent perceive the external world via sensors, and recognising the objects therein contained. The latter problem is inherently sub-symbolic as it requires acquiring, processing, and fusing raw data coming from the sensors.

2.2 Data Science

This section contains contributions from the following works of ours: [CCDO19, CCO20, CSOC20]

Data science (DS) is a relatively young discipline laying at the intersection among AI, Statistics, CS and SE. It essentially deals with the extraction of relevant information out of data, and, in particular, with the *data-driven* creation of *predictive* models of real world phenomena. Thanks to its focus on predictive models, DS is applied to virtually all statistical sciences, ranging from physics to law, stepping through biology, healthcare, or finance. In all such scenarios, the reliance on real-world data is quintessential to tune such predictive models in such a way to make them adhere to reality.

Data science can be described w.r.t. two major perspectives, here referred to as the *scientific* and the *engineering* perspectives on DS. The scientific perspective focuses on the central aspect of DS – namely, data – and on *what* algorithms, workflows, and practices can be exploited to process data to serve specific analytic purposes, in a sound way. The engineering perspective focuses on *how* to make such processing efficient and effective, in spite of the large volumes, and the wide variety of data required to this purpose. Within this scope, another relevant concern is the velocity at which data is produced, and processed information is consumed.

The Scientific Perspective. As a science, DS studies the many means one can exploit to *(i)* let a software agent learn new behaviours from examples, which would otherwise be hard to encode for human developers (e.g. handwritten text recognition), *(ii)* automatically recognise patterns of similar objects given a number of examples (e.g. face detection), *(iii)* detect recurrent patterns in data, even in lack of prior examples (e.g. customer profiling), *(iv)* predict the future evolution of a phenomenon given its historical data (e.g. stock performance predictor), *(v)* simulate the dynamics of complex phenomena (e.g. weather forecasting), *(vi)* figure out the mathematical relation biding two or more variables, from a number of samples (e.g. studying estate market prices), *(vii)* fuse data coming from different sources to infer unobservable measures (e.g. indoor localization), etc. other than, of course the theories and practices to assess and increase the predictive performance of all such models. In doing so, DS borrows countless algorithms, methods, and techniques from disparate fields, including but not limited to machine learning (there including supervised, unsupervised, and reinforcement learning), data mining, Bayesian inference, statistics, etc.

Despite the plethora of algorithms and methods which lay nowadays under the DS umbrella, a concise overview of the discipline can be outlined in terms of *tasks*. Several algorithms can be used in DS to perform a well-established pool of data-analytics tasks having a clear knowledge-related goal. Most common tasks in DS are for instance:

function fitting (a.k.a. classification or regression) — i.e. the *supervised* learning task of inferring the input-output relation among a number of samples, to be later able to estimate likely outputs for novel, unseen inputs;

clustering — i.e. the *unsupervised* learning task of finding similar groups of instances in a dataset – according to a given notion of *similarity* or distance –, to be later able to classify novel instances according to some group;

anomaly detection — i.e. the *unsupervised* learning task of tuning an algorithm to discern "normal" situations from exceptional ones, provided that some historical data is available, to later be able to detect the latter;

filtering (resp. **smoothing** or **forecasting**) — i.e. the Bayesian task of estimating the unknown *current* (resp. *past* or *future*) state of a system given a sample of observations capturing the evolution of a number of variables which should depend on that state;

most likely explanation — like the above, but focussing on the most likely *sequence* of states a system has traversed.

Most of these tasks may be implemented in several ways and by several algorithms. For instance, function fitting alone may be realised using neural networks, (generalised) linear models, support vector machines, decision trees, and many others. Notably, the same algorithm could be used to implement several tasks—e.g. neural networks can be exploited to perform both classification and regression tasks, as well as anomaly detection.

For all such tasks, two major phases are commonly identified in DS, namely *training* (a.k.a. learning, or fitting) and *usage* (a.k.a. inference). The first phase (training) commonly occurs behind the scenes, and it is led by *data scientists* – i.e. human beings –, despite involving semi-automated workflows. The second phase (usage) is commonly what AI consumers use and deal with. During training, the most adequate algorithm is selected for the data and the task at hand, and it is then trained on data, producing a predictive *model* which, hopefully, is predictive enough to be later used on novel data. Users may then exploit the model by feeding it with novel data, to draw predictions. Predictions, in turn, may be presented to the users as recommendations, decisions, or outcomes.

Notably, regardless of their technical details, all the data-analytics tasks above may leverage on a number of lower-level ancillary activities which are orthogonal w.r.t. the choice of the particular implementing algorithm, as they involve routine operations, assessment procedures, or best practices which are commonly executed either before or after the data-analytics task itself. Examples of such kinds of activities are, for instance:

feature engineering — i.e. a whole class of pre-processing techniques – such normalising numeric data into predefined intervals, changing the way data is encoded, or creating new data from the available one – which may be used to improve or transform the available data, in order to improve the performance of the data analytics task;

dimensionality reduction — i.e. a whole class of data manipulation techniques aimed at selecting the most relevant attributes of data for a given data-analytics task (before running the task)—such as principal component analysis;

model assessment i.e. a whole class of statistical methods, algorithms, and practices to assess the performance of the data-analytic task in a principle way—such as for instance supervised learning metrics (e.g. accuracy, or mean-squared error) or procedures such as cross-validation or test set separation;

model selection i.e. a number of strategies, approaches, and practices to let data scientists select the best predictive model when multiple options fit the data and the situation at hand—e.g. by leveraging on *cross-validation*, possibly in combination with a grid search strategy to identify the most adequate type of predictor.

The Engineering Perspective. As a field of engineering, DS deals with the design and implementation of robust software and hardware architectures, which support the *scalable* execution of the aforementioned data-analytic tasks over the so-called *big data*. For this reason, the engineering perspective of DS is also known as the field of *big data processing*.

The exploitation of parallel or distributed solutions to speed up the data processing workflows is the most relevant object of study under the engineering perspective of DS. Along this line, there are two broad sorts of situations which is worth mentioning, namely: *on-line* or *off-line* data processing. They correspond to as many major approaches to big data processing, namely *stream* processing and *parallel/distributed* computing.

On-line data processing deals with the need of processing data as soon as it is produced, without any intermediate accumulation phase. The outcomes of on-line data processing must be consumed in useful time—hence the need to process it quickly, on the fly. Along this line, the notion of data *stream* – that is, a possibly *unlimited* sequence of data to be *lazily* and reactively processed – is fundamental as it supports the processing of large amounts of data without requiring them to be simultaneously stored in memory—therefore enabling great scalability.

Off-line data processing deals with the need of analysing data statistically, therefore requiring as much data as possible. A data accumulation phase is commonly the underlying implicit assumption for off-line data processing. Accordingly, the focus here is on speeding up – through parallelisation – the data processing workflows which would otherwise require too much computational time or power. There, parallelisation may occur on either on multiple cores of the same machine, or multiple *distributed* machines.

Chapter 3

Representing Data and Knowledge

Representation deals with the expression of information to make it understandable and manipulable by agents—be they computational or humans. From a philosophical perspective, there are two major premises to any well-funded discussion on representation.

First, both computational and human agents operate (i.e. compute or think) upon *representations* of relevant aspects of the reality—and representations are everything an agent may ever hope to manipulate. *Noumena* – i.e. what things actually are – are not accessible directly, but rather via *perception*. Perception implies consuming some input data, which must in turn be represented, to enable further processing. So, agents always deal with *phenomena* – i.e. how things appear –, hoping that the corresponding *noumena* are reflected with sufficient precision. (Of course, to reach a true understanding about a particular noumenon, several related phenomena should be observed, but this particular aspect is addressed in the following chapters.)

Second, representations are manifold and of different sorts, and they may focus on particular aspects of the phenomenon being represented. In other words, whenever an agent is dealing with some information, it is actually dealing with a particular representation of some underlying concept, despite many others could be available. Furthermore, representations are never good or bad per se, but rather more or less adequate to the agent exploiting them and to the task it is performing. So, by whom information must be consumed, and to serve what purpose, is a relevant concern in deciding which representations are more adequate.

Despite being rooted into deep and long-standing philosophical discussions, such premises are here reported serving a practical purpose. Indeed, they synthesise the underlying mindset tying this chapter and the following ones together: the particular choice of a particular means to representation simultaneously enables and constrains the kinds of possible processing information may be subject to—and this in turns conditions any subsequent design choice.

Accordingly, within the scope of this chapter we discuss the representation of particular sorts of information, namely either *data* or *knowledge*. We consider as data any *raw* information attained by *sampling* some phenomenon or situation from reality. Data by itself simply describes the phenomenon / situation, yet it is hard to exploit and transfer directly, because of its granularity and volume. Conversely, we consider as knowledge any coarse-grained piece of information describing entire classes of phenomena or situations, in a concise and reusable (i.e. predictive) way. Differently from data, knowledge can be applied to unseen phenomena or situations, or transferred to agents which have not experienced any such phenomena / situations explicitly.

Be it devoted to data or knowledge, each representation comes with pros and cons, simplifying the expression of some aspects of the information being represented, while complicating the expression of others. Indeed, a lot of effort in DS is devoted to the engineering of the best representation for the data at hand, to maximise the effectiveness of any subsequent data-processing task.

The means to represent data and knowledge are manifold and too many to count. However, at the meta-level, we can categorise representations means as either *symbolic* or *sub-symbolic*. While the two means are essentially interchangeable – other than mutually convertible – when they represent data, they lead to profoundly different ways of representing knowledge. Indeed, while symbolic knowledge is both machine- and human-interpretable, sub-symbolic is mostly machine-interpretable, and is therefore treated by human beings as a black box in the general case.

Along this line, in the reminder of this chapter we focus on logic – as the most prominent approach to *symbolic* representation –, and vectors, matrices, or tensors of real numbers—as the most prominent approach to *sub-symbolic* representation. We show analogies and differences among such approaches to representation, eliciting the pros and cons of both, and, in particular, their differences among the interpretability perspective.

3.1 Symbolic Knowledge Representation

(Symbolic) Knowledge representation (KR) has always been regarded as a key issue since the early days of AI, as no intelligence can exist without knowledge, and no computation can occur in lack of representation.

Here we discuss the language of FOL as a means for representing symbolic information. We choose FOL as it is quite general, and the other approaches can be described by either constraining or loosening the definition of FOL.

⟨Formula⟩	:=	⟨Clause⟩ $\mid$ ⟨Quantifier⟩⟨Formula⟩
⟨Quantifier⟩	:=	‘$\forall$’⟨Variable⟩ $\mid$ ‘$\exists$’⟨Variable⟩
⟨Clause⟩	:=	⟨Literal⟩ $\mid$ ‘(’⟨Formula⟩⟨Connective⟩⟨Formula⟩‘)’
⟨Connective⟩	:=	‘$\wedge$’ $\mid$ ‘$\vee$’ $\mid$ ‘$\rightarrow$’ $\mid$ ‘$\leftrightarrow$’ $\mid$ ‘$=$’
⟨Literal⟩	:=	⟨Predicate⟩ $\mid$ ‘$\neg$’⟨Predicate⟩
⟨Predicate⟩	:=	‘$\top$’ $\mid$ ‘$\bot$’ $\mid$ ⟨Predication⟩ $\mid$ ⟨Predication⟩‘(’⟨Arguments⟩‘)’
⟨Predication⟩	:=	$p_1 \mid p_2 \mid p_3 \mid \ldots$
⟨Arguments⟩	:=	⟨Term⟩ $\mid$ ⟨Term⟩‘,’⟨Arguments⟩
⟨Term⟩	:=	⟨Variable⟩ $\mid$ ⟨Structure⟩ $\mid$ ⟨Constant⟩
⟨Variable⟩	:=	$X_1 \mid X_2 \mid X_3 \mid \ldots$
⟨Structure⟩	:=	⟨Functor⟩‘(’⟨Arguments⟩‘)’
⟨Functor⟩	:=	$\mathtt{f}_1 \mid \mathtt{f}_2 \mid \mathtt{f}_3 \mid \ldots$
⟨Constant⟩	:=	⟨Functor⟩ $\mid$ ⟨Number⟩ $\mid$ ⟨Boolean⟩
⟨Number⟩	:=	$\mathbb{R}$

Table 3.1: Context-free grammar for FOL. Sans-serif words among angular brackets denote non-terminal symbols, whereas symbols among single apices denote terminal symbols

3.1.1 First Order Logic (FOL)

First order logic (FOL) [Smu68] is a general-purpose logic which can be used to represent knowledge symbolically, in a very flexible way. More precisely, it allows both human and computational agents to express (i.e. write) the properties of – and the relations among – a set of entities constituting the *domain of the discourse*, via one or more *formulæ*—and, possibly, to reason over such formulæ by drawing inferences.

In table 3.1 the syntax of FOL is formally defined via a context-free grammar. Informally, the syntax for the general FOL formula is defined over the assumption that there exist:

- a number of constant symbols, including: a number of *functors*, denoted by monospaced symbols such as $\mathtt{f}_1, \mathtt{f}_2, \ldots$, and all real numbers;
- a number of *predicate symbols* (a.k.a. predications), denoted by italic symbols starting with a lower case letter, such as $p_1, p_2, \ldots$;
- a number of *variables*, denoted by italic symbols starting with a capital letter, such as $X_1, X_2, \ldots$.

Under such assumption a FOL formula is any expression composed by a list of quantified variables, followed by a number of *literals*, i.e. *predicates* which may or may not be prefixed by the negation operator ($\neg$)—in which case would be called negated. Each predicate consists of a predicate symbol, possibly applied to one or more *terms*. More precisely, each predicate may carry $N \geq 0$ terms. When this is the case, the predicate is said N-ary (meaning that its arity, or amount of arguments, is N). Terms may be of three sorts, namely *constants*, *structures*[1], or *variables*. Constants represent the many entities from the domain of the discourse. In particular, each constant references a different entity. Structures are combinations of one or more entities via one *functor*[2]. Similarly to predicates, structures may carry $M \geq 0$ terms. When this is the case, the structure is said M-ary as well. Being containers of terms, structures enable the creation of arbitrarily complex data structures combining several entities from the domain of the discourse, and treating them as a whole. Finally, variables are placeholders for unknown terms—i.e. for entities or groups of entities.

Predicates and terms are very flexible tools to represent knowledge. While terms can be used to represent or reference either entities or groups of entities from the domain of the discourse, predicates can be used to represent relations among those entities, or the properties of each single entity. There, the domain of the discourse $\mathbb{D}$ [Bla08] is the set of all relevant entities which should be represented in FOL to amenable of formal treatment, in a particular scenario. Should we use FOL to treat arithmetic, $\mathbb{D}$ would include the set of *natural* numbers—i.e. a symbol for each natural number. Should we treat calculus, $\mathbb{D}$ would include the set of *real* numbers. Should we treat kinship relationships, $\mathbb{D}$ would include a symbol for each person taken into account.

Concerning predicates, let us denote by p/N the N-ary predicate whose predicate symbol is p and whose arity is N. When $N \geq 2$, the predicate represents one or more items from the relation $p \subseteq \mathbb{D} \times \ldots \times \mathbb{D}$. So, for instance, the expression $p(t_1, \ldots, t_N)$, where all t_i are non-variable terms, denotes that the N-uple $(t_1, \ldots, t_N)$ is part of the N-ary relation subtended by p—or that, in other words, the relation N-ary relation p ties the entities $t_1, \ldots, t_N$ together. Similarly, the expression $\forall X_i \; p(t_1, \ldots, X_i, \ldots, t_N)$, where X_i is a variable, denotes a situation where, for each entity X_i in $\mathbb{D}$, the relation N-ary relation p ties the entities $t_1, \ldots, X_i, \ldots, t_N$ together. Dually, the expression $\exists X_i \; p(t_1, \ldots, X_i, \ldots, t_N)$ denotes an item of the N-ary relation p whose i^{th} item is unknown, or, in other words, an item where the first argument is t_1, the second argument is t_2, ..., and the last argument is t_N, while the i^{th} argument is arbitrary. Conversely, when $N = 1$, the predicate represents one or more items from the set $p \subseteq \mathbb{D}$. So, for

[1]structures are also (and most commonly) known as "functions" into the CL literature. However, functions in CL denote data structures rather than associations among a domain and a co-domain, as they are commonly intended. Thus, to avoid ambiguity, we choose to call then "structures" instead.

[2]functors are also known as "function symbols" in the CL literature

instance, the expression $p(t)$, where t is a non-variable term, denotes a situation where t is an item of the set subtended by p—or that, in other words, the property p holds for the entity t. Similarly, the expression $\forall X\ p(X)$, denotes a situation where all items in $\mathbb{D}$ are items of the set p as well—or that, in other words, the property p holds for all entities in $\mathbb{D}$. Dually, the expression $\exists X\ p(X)$, denotes a situation where some item in $\mathbb{D}$ is in p as well. Finally, when $N = 0$, the predicate p represents a Boolean proposition which may or may not be true. Notably, the predicate $\top$ is always true, by construction, whereas the predicate $\bot$ is always false.

Concerning non-variable terms, let us denote by $\mathtt{f}/M$ the M-ary term whose functor is `f` and whose arity is M. When $M \geq 0$, the term is a constant and it represents some entity from $\mathbb{D}$. When $M \geq 1$, the term is a structure – i.e. a named and ordered group of terms – and it represents a complex or composite entity from $\mathbb{D}$. The actual interpretation of a structure really depends on the scenario at hand. So, for instance, in the arithmetic domain, it is possible to represent natural numbers by mimicking the Peano axioms[3] via a unary structure – e.g. `s`, for *successor* – and a constant – e.g. `z`, for *zero* – as follows: `z` represents 0, `s(z)` represents 1, `s(s(z))` represents 2, etc. Under this representation, each natural number (except `z`) is composed by its predecessor, and the successor functor $\mathtt{s}/1$.

Structures as composite entities. Structures may be used in logic to represent composite entities. Such composite entities may either be of fixed size or of variable size.

A fixed-size composite entity made up of M sub-entities may be represented in FOL via a M-ary structure. For instance, one may represent a person in terms of first name, last name, and birthdate. In that case $M = 3$, and an adequate functor is '`person`':

$$\mathtt{person}(\mathtt{adam},\ \mathtt{smith},\ \mathtt{date}(1723,\ \mathtt{june},\ 5))$$

The underlying assumption here is that dates are represented as ternary structures as well.

A variable-size composite entity, in turn, may be made up of an unknown amount of sub-entities. Furthermore, two different composite entities of the same sort may be of different sizes. Consider for instances two different journeys on a map: one may involve 3 cities, and the other may involve 4 cities, yet both can be represented by *lists* of cities to be visited in a row.

Lists – and, more generally, data structures – can be represented in FOL via ad-hoc fixed-size structures, to be used recursively. In particular, a common con-

[3]`https://www.britannica.com/science/Peano-axioms`

vention is to represent *singly linked* lists of entities using:

- a binary structure, denoting element–successor couples, e.g. `cons`/2 or ./2,
- a constant, denoting the termination of the list, e.g. `nil` or [].

So, for instance, a journey from Rome to Milan, stepping through Florence and Bologna may be represented as follows:

$$\texttt{cons}(\texttt{rome},\ \texttt{cons}(\texttt{florence},\ \texttt{cons}(\texttt{bologna},\ \texttt{cons}(\texttt{milan},\ \texttt{nil})))) \tag{3.1}$$

whereas a journey from Rome to Naples would be as simple as:

$$\texttt{cons}(\texttt{rome},\ \texttt{cons}(\texttt{naples},\ \texttt{nil}))$$

In both cases, cities are represented by constants, whereas lists of cities are attained by combining cities into data structures—i.e. by *recursively* wrapping cities via the `cons`/2 functor, and by exploiting the constant `nil` to conclude the list.

It is worth to be mentioned that, a more practical and common notation involves the exploitation of ./2 and [] instead of `cons`/2 and `nil`, respectively, where ./2 is usually written as an *infix* symbol. With this notation, the path from eq. (3.1), could be written as:

$$\texttt{rome}\ .\ \texttt{florence}\ .\ \texttt{bologna}\ .\ \texttt{milan}\ .\ []$$

or, equivalently:

$$[\texttt{rome},\ \texttt{florence},\ \texttt{bologna},\ \texttt{milan}]$$

Knowledge Bases. From a knowledge representation perspective, knowledge bases (KB) (a.k.a. *theories*) are sets of related FOL formulæ concerning the same domain of the discourse. We denote theories as lists of dot-terminated formulæ.

For instance, a simple KB describing natural numbers may be defined as follows:

$$\begin{gathered} natural(\texttt{z}). \\ \forall X\ natural(X) \rightarrow natural(\texttt{s}(X)). \end{gathered} \tag{3.2}$$

There, the KB is composed by two formulæ, and it aims to define the set of natural numbers by means of the unary predicate *natural*/1, the unary structure `s`/1, and the constant `z`. More precisely, the first formula states that the constant `z` is included into the set of natural numbers, by construction, whereas the second one states that, whenever some object X is in the set of natural numbers, then object $\texttt{s}(X)$ is in the same set as well. By recursively applying that formula, one may express any natural number in Peano notation.

Intensional vs. Extensional. The recursive definition of natural numbers from eq. (3.2) is also interesting because it exemplifies the difference among *extensional* and *intensional* definitions.

In logic, one may define concepts – i.e. describe data – either extensionally or intensionally. Extensional definitions are *direct* representation of data. In the particular case of FOL, this implies defining a relation or set by explicitly mentioning the entities it involves. The $natural(\mathtt{z})$ formula from eq. (3.2) is a particular case of *extensional* definition of the symbol $\mathtt{z}$ as a natural number. In other words, it partially defines the *natural* set by specifying some of its items. Conversely, intensional definitions are *indirect* representations of data. In the particular case of FOL, this implies defining a relation or set by describing its elements via other relations or sets. The $\forall X\ natural(X) \rightarrow natural(\mathtt{s}(X))$ formula from eq. (3.2) is a particular case of *intensional* definition of the any symbol of the form $\mathtt{s}(X)$ as a natural number, provided that X is a natural number as well.

Notice that the focus here is *not* on recursion. Intentional definitions must not necessarily be recursive. For instance, one may intensionally define the $child/2$ relation via the $parent/2$ relation as follows:

$$\forall X\ \forall Y\ parent(X, Y) \rightarrow child(Y, X).$$

Yet, recursive intensional predicates are very expressive and powerful, as they enable the description of infinite sets via a finite (and commonly small) amount of formulæ.

Herbrand and its ground. Variables play a fundamental role in intensional KR, as they allow referencing unknown entities and tie them together via either predicates or structures. However, there exists situations – described later in this thesis – where the presence of variables may be troublesome. Accordingly, here we provide a number of definitions related to variable-free FOL formulæ and KB.

A term is considered *ground* if and only if *(i)* it is a constant, or *(ii)* it is a structure ant it is only composed by constant or ground arguments. In other words, a term if it contains no variable, not even recursively. A predicate is ground it any term therein contained is ground as well. A formula is ground if it only contains ground predicates, and a KB is ground if it only contains ground formulæ.

We call *Herbrand universe* the set of all possible ground terms, denoted by $\mathcal{H}$. In other words, $\mathcal{H}$ is the set of all possible representations of all entities in the domain of the discourse. Given a set of constants and functor symbols, $\mathcal{H}$ can be recursively defined as the set containing: *(i)* all possible constants, and *(ii)* all structures attained by applying all possible M-ary functors to each possible M-uple of items in $\mathcal{H}$.

The Herbrand universe may easily become infinite. A single functor of arity greater than 0 – say, `f` – plus a single constant – say, `x` – are sufficient to create an infinite Herbrand universe, as the functor may be recursively applied to the constant, infinitely many times—i.e. $\mathcal{H} = \{\mathtt{x}, \mathtt{f(x)}, \mathtt{f(f(x))}, \ldots\}$.

3.1.2 Representation Engineering

When handling knowledge in practice, the particular way knowledge is modelled is quintessential for computational systems to be effective. Of course, any particular modelling is better suited to support some sorts of algorithms while it may make the exploitation of other algorithms cumbersome. In other words, the particular shape of predicates and structures may be chosen in manifold ways, depending on the nature of the data at hand, and on the computations that designers are expecting for that data.

Here we briefly examinate the two extremes in a spectrum of possibilities, with the purpose of discussing how each choice in KR may come with both pros and cons—which must therefore be engineered.

We rely on two running examples, namely the "ties of kinship" example – mimicking a simple scenario where the ties of kinship among a number of people must be represented –, and the "Iris" example, where data about a number of Iris flowers are collected. For both of them, we discuss possibilities in KR strategies.

Representing relational data. We here consider a simple scenario where the ties of kinship among a number of people must be represented via FOL. We take Abraham's family tree from the Genesis as an example.

A natural way to represent a family tree is by using constants to represent people, while extensively representing a minimal pool of relations, and intensively representing any other relation—in both cases, via predicates. For instance, we may choose to extensively represent parenthood relations among couples of people, other than the gender of each person. Other kinds of relations could be represented intensively. For example:

$$\begin{array}{rcl}
parent(\mathtt{abraham}, \mathtt{isaac}). & & male(\mathtt{abraham}). \\
parent(\mathtt{sarah}, \mathtt{isaac}). & & female(\mathtt{sarah}). \\
parent(\mathtt{isaac}, \mathtt{jacob}). & & male(\mathtt{isaac}). \\
parent(\mathtt{rebekah}, \mathtt{jacob}). & & female(\mathtt{rebekah}). \\
\ldots & & male(\mathtt{jacob}). \\
\forall X\ \forall Y\ parent(X,Y) & \rightarrow & child(Y,X). \\
\forall X\ \forall Y\ parent(X,Y) \wedge male(X) & \rightarrow & father(X,Y). \\
\forall X\ \forall Y\ parent(X,Y) \wedge female(X) & \rightarrow & mother(X,Y). \\
\forall X\ \forall Y\ \exists Z\ parent(X,Z) \wedge parent(Z,Y) & \rightarrow & grandparent(X,Y).
\end{array} \tag{3.3}$$

This approach to KR is particularly adequate to describe situations involving several entities and many relations or properties, but where, however, each predicate only spans through a few entities, and most entities are not involved in all relations or properties. In other words, this approach is well suited to represent *heterogeneous* data.

Representing propositional data. We here consider a simple scenario where the well-known Iris dataset[4] is represented via FOL. Notably, the Iris dataset is a collection of 150 individuals of the Iris flower. For each exemplary, 4 numeric input features – petal and sepal width and length – are recorded, other than a class label—i.e. which particular sort of Iris plant the exemplary has been classified as. There are three particular sub-sorts of Iris in this data set – namely, Iris-Setosa, -Virginica, and -Versicolor –, and the 150 examples are evenly distributed among them—i.e. there are 50 instances for each class.

The Iris dataset essentially consists of a bi-dimensional 150×5 table, where each row corresponds to an exemplary, each column corresponds to a relevant *feature*, and each cell carries the value of a particular feature for a particular exemplary. A natural way to represent N – e.g. 150 – records of equals size M – e.g. 5 – is by leveraging on N predicates, all having the same arity M, and the same functor—e.g. $iris$. There, each predicate *extensionally* represents an instance of the same M-ary relation – e.g. $iris$ –, and the j^{th} argument of each predicate carries the value of the j^{th} feature for that instance—according to some predefined ordering of features. Thus, the values corresponding to numeric features can be represented in FOL by numeric constants, while the values corresponding to the class feature could be represented by ad-hoc constants. So, a KB describing the Iris dataset in FOL according to this may look as follows:

$$\begin{gathered} iris(5.1,\ 3.5,\ 1.4,\ 0.2,\ \texttt{setosa}). \\ \vdots \\ iris(7.0,\ 3.2,\ 4.7,\ 1.4,\ \texttt{versicolor}). \\ \vdots \\ iris(6.3,\ 3.3,\ 6.0,\ 2.5,\ \texttt{virginica}). \end{gathered} \tag{3.4}$$

where the predefined ordering of features is: *(i)* sepal length, *(ii)* sepal width, *(iii)* petal length, *(iv)* petal width, and *(v)* class.

This approach to KR is particularly adequate to describe situations where the same amount and sorts of fields are available for each datum, thus making the whole dataset suitably described by an $N \times M$ table—and, therefore, therefore extensively represented as an M-ary relation having with N instances. In other

[4] `https://archive.ics.uci.edu/ml/datasets/iris`

words, this approach is well suited to represent *homogeneous* data.

Comparison. Both approaches to KR come with both pros and cons, and they are, at least in principle, interchangeable—meaning that, conversions may be performed from data represented via any of the two approaches into the other. Here, we simply highlight how the effectiveness of KR heavily depends on the particular computation to be performed.

There are two kinds of activities one may use as benchmarks to assess the limits of each approach to KR, namely: *(i)* enumerating all the individuals involved into a KB and all the features describing them, and *(ii)* adding one new feature to the KB, updating all involved individuals accordingly.

On the one side, in the propositional approach, the amount of individual is equal to the amount on predicates, whereas the amount of features is equal to the arity of all predicates. So enumeration of both individuals and features is straightforward. Conversely, in the relational approach, individuals should be enumerated by stepping through all predicates, and removing duplicates; whereas the enumeration of all features may require a lot of computations—as the intensionally represented features should be made explicit.

On the other side, adding a new feature to a KB represented via heterogeneous approach is straightforward. It just requires the novel predicates to be added to the KB. Conversely, in the propositional approach, the same operation would require the *whole* KB to be rewritten—to let each predicate carry one more feature.

3.1.3 Relevant Subsets of FOL

Historically, most KR formalisms and technologies have been designed around either sub-sets or applications of the *first order logic* (FOL). Consider for instance, *deductive databases* [GR68], *description logics* [Baa03], *ontologies* [Cim06], *Horn* logic [McN77], *higher-order* logic [VBD01], just to name a few.

Many kinds of logic-based knowledge representation systems have been proposed over the years, mostly relying on FOL – either by restricting or extending it –, e.g. on description logics and modal logics, which have been used to represent, for instance, terminological knowledge and time-dependent or subjective knowledge.

Ontologies and Description Logics. Early KR formalisms, such as *semantic networks* and *frames* [Sow91], mostly aimed at providing a structured representation of information. For this reason, description logics are characterised by several restrictions w.r.t. to FOL Applications range from reasoning with database schemas and queries [AFWZ02] to *ontology languages* such as OIL, DAML+OIL and OWL [Hor05]—always keeping in mind that not only the key inference problems should be decidable, but also that the decision procedures should be implemented *efficiently*.

Ontology-based approaches are popular because of their basic goal—a common understanding of some domain that can be shared between people and application systems. At the same time, it should be understood that the general concepts and relations of a top-level ontology can rarely accommodate all of the systems peculiarities [VEBB$^+$08, Val05].

A number of systems based on description logics have been developed – e.g. [CH94, MH03] – in diverse application domains, such as natural language processing, configuration of technical systems, software information systems, optimising queries to databases, planning.

Horn Logic. Horn logic is a notable subset of FOL, characterised by a good trade-off among theoretical expressiveness, and practical tractability [Mak87].

Horn logic is designed around the notion of *Horn clause* [Hor51]. Horn clauses are FOL formulæ having no quantifiers, and consisting of:

- a disjunction of predicates, where only at most one literal is non-negated:
$$\neg b_1 \vee \ldots \vee \neg b_n \vee h$$
- or, equivalently (applying De Morgan rules), a disjunction among a predicate and a negated conjunction of predicates:
$$\neg(b_1 \wedge \ldots \wedge b_n) \vee h$$
- or, equivalently (applying the equivalence $\neg X \vee Y \equiv X \rightarrow Y$), an implication having a single predicate as post-condition and a conjunction of predicates as pre-condition:
$$b_1 \wedge \ldots \wedge b_n \rightarrow h$$
- often conveniently written as:
$$h \leftarrow b_1,\ \ldots,\ b_n \tag{3.5}$$

where $\leftarrow$ denotes logic implication from right to left, commas denote logic conjunction, and all b_i, as well as h, are predicates of arbitrary arity, possibly carrying FOL terms of any sort—i.e. variables, constants, or structures. By looking at eq. (3.5), it should be evident why h is often called *head* (of the clause), while the conjunction $(b_1, \ldots, b_n)$ is often called *body* (of the clause). Quantification of variables is omitted, as all variables possibly occurring in the head are assumed to be *universally* quantified, whereas all other variables possibly occurring in the body (and not in the head) are assumed to be *existentially* quantified.

So, essentially, Horn logic is a very restricted subset of FOL where:

- formulæ are reduced to clauses, as they can only contain predicates, conjunctions, and a single implication operator, therefore
- operators such as $\vee$, $\leftrightarrow$, or $\neg$ cannot be used,
- variables are implicitly quantified, and
- terms work as in FOL (there including the definition of "ground term" and "Herbrand universe").

Similarly to FOL, Horn logic KB consist of sets of Horn clauses.

Despite being very restrictive in theory, the lack of basic operators such as $\vee$, $=$, or $\neg$ can be circumvented in practice, via *meta-predicates*—i.e. predicates accepting other predicates as arguments. Circumvention in these cases steps through a smart trick: by letting the set of admissible functors *include* the set of possible predicate symbols, one may enable the representation of predicates via terms. So, a meta-predicate can be described as an ordinary predicate, accepting terms as arguments, and considering its arguments as predicates. However, these aspects are covered in chapter 5.

It is worth to be noted that Horn clauses can be read under both a *logic* and a *procedural* perspective [vEK76]. Under a logic perspective, Horn clauses are bare implications, which can be used to define relations or sets, as in FOL. Under a procedural perspective, any Horn clause states that "to prove h, one should prove all $b_1, \ldots, b_n$ first". Along this line, when Horn clauses are exploited in practice, they are commonly referred to as

facts when their body consist of just the $\top$ predicate:

$$h \leftarrow \top \quad \text{or simply} \quad h$$

stating that h is known to be true (as it requires nothing to be proven first),

goals (a.k.a.**directives**) when their head consist of just the $\perp$ predicate (or, equivalently, when the head is missing):

$$\perp \leftarrow b_1,\ \ldots,\ b_n \quad \text{or simply} \quad \leftarrow b_1,\ \ldots,\ b_n$$

stating that predicates $b_1, \ldots, b_n$ should be all proven,

rules otherwise, i.e. when both the head and the body involve arbitrary predicates.

It is worth to be mentioned that facts are particular case of rules. Indeed, both facts and rules are also known as *definite* clauses.

3.2 Sub-symbolic Data Representation

Symbolic KR approaches, such as FOL and its subsets, represent both data and knowledge uniformly—meaning that they provide a common language capable of representing both. The same statement does not hold for sub-symbolic approaches, which commonly represent data as (possibly multi-dimensional) *arrays* (e.g. vectors, matrices, or tensors) of real numbers, and knowledge as functions over such data.

Despite numbers are technically symbols as well, we cannot consider arrays and their functions of as symbolic KR means. Indeed, according to [vG90], to be considered as symbolic, KR approaches should: *(i)* involve a set of symbols, *(ii)* which can be combined (e.g. concatenated) in possibly infinite ways, following precise grammatical rules, and *(iii)* where both elementary symbols and any admissible combination of them can be assigned with meaning—i.e. each symbol can be mapped into some entity from the domain of the discourse. In this section we discuss how sub-symbolic approaches are characterised by the frequent violation of items *(ii)* and *(iii)*.

Vectors, matrices, tensors. Multi-dimensional arrays are the basic brick of sub-symbolic data representation. More formally, a D-order array consists of an ordered container of real numbers, where D denotes the amount of indices required to locate each single item into the array. The i^{th} index of the array is assumed to range through the interval $1, \ldots, d_i$, so that the whole dimension of the array – i.e. the total amount of numbers therein contained – is $d_1 \times \ldots \times d_D$. In what follows, we may abuse the notation by referring to 1-order arrays as *vectors*, 2-order array as *matrices*, and higher-order arrays as *tensors*. Along this line, we may also denote by $\mathbb{R}^n$ the set of n-dimensional vectors, by $\mathbb{R}^{n \times m}$ the set of $(n \times m)$-dimensional matrices, and by $\mathbb{R}^{d_1 \times \ldots \times d_D}$ the set of $(d_1 \times \ldots \times d_D)$-dimensional tensors.

In any given sub-symbolic data-representation task leveraging upon arrays, information may be carried by both:

- the actual numbers contained into the array, and
- their location into the array itself.

In practice, the actual dimensions ($d_1 \times \ldots \times d_D$) of the array play a central role as well. Indeed, as further discussed in chapter 4, sub-symbolic data processing is commonly tailored on arrays of *fixed* sizes—meaning that the actual values of $d_1, \ldots, d_D$ are chosen at design time and never changed after that. For this reason, we define sub-symbolic data representation as the task of expressing data in the form of *rigid* arrays of *numbers*. Notably, such a task is *extensional* by construction, as information can only be explicitly represented.

Local vs. Distributed. An important distinction, when data is represented in the form of numeric arrays, is about whether the representation is *local* or *distributed* [vG90]. In local representations, each single number into the array is characterised by a well-delimited meaning—i.e. it is measuring or describing a clearly identifiable concept from the domain of the discourse. Conversely, in distributed representations, each single item of the array is nearly meaningless, unless it is considered along with its neighbourhood—i.e. any other item which is “close” in the indexing space of the array, according to some given notion of closeness. So, while in local representations the location of each number in the array is quite negligible, in distributed representations it is of paramount importance.

Consider for instance the Iris dataset from section 3.1.2: it is a tabular dataset where each datum can be considered as a 5-dimensional vector. There, each component of the vector is informative *per se*: it may describe e.g. the petal/sepal length/width. Conversely, consider a dataset of black/white images whose resolution is $w \times h$. There, each image can be represented as a $h \times w$ matrix of numbers in the range $[0, 1]$, where each location represents a pixel and the corresponding brightness. The single pixel carries very small information when considered alone, whereas groups of contiguous pixel may describe details which are relevant for image processing.

Feature Engineering. Of course, not all data is both rigid and numeric in nature. So, to fit this paradigm, data scientists designed a plethora of conversion methods to transform data from various forms (e.g. possibly non-rigid or non-numeric, when not both) into rigid arrays of numbers. In particular, when raw data is very flexible (i.e. variable in size) and very distributed, a common method consists of computing the so-called *embeddings*, i.e. fixed-size arrays synthesising the information contained into the raw data. All such methods lay under the *feature engineering* umbrella.

The ideal situation, under a data representation perspective, is when data is in *tabular* form, i.e. N instances and M features, and all features only involve numeric values. There, each instance is naturally described by a M-dimensional vector, while the whole dataset is described by an $N \times M$ matrix. However, in practice, only rarely raw data fits the rigid and numeric paradigm since the very beginning. More commonly, raw data may diverge from the paradigm in several ways—possibly, simultaneously. When this is the case, a number of transformations can be applied to the data to make it converge to the paradigm.

Non-numeric features. A dataset may involve non-numeric features, even when of tabular form. When this is the case, each single non-numeric feature may be transformed into numeric by applying a transformation to each value. The most adequate transformation heavily depends on the domain of the feature itself:

boolean features may be trivially converted into numbers via the $\{\texttt{false} \mapsto 0, \texttt{true} \mapsto 1\}$ encoding;

ordinal features may be trivially converted into natural numbers reflecting the same ordering;

categorical features may be converted into boolean features via the one-hot encoding[5];

structured features having a **fixed** structure (e.g. dates or timestamps) can be decomposed into their components;

while other situations may fit the cases below.

Variable-size data. A dataset may involve data of variable size. Consider for instance time series (e.g. samples of some phenomenon over time), or free text, or graph-like information (e.g. friendships on social networks, citations in papers). There, despite each single instance of the dataset can be trivially translated into an array of numbers of some size, any two different instances from the same dataset may have different sizes and internal structures.

For instance, time series can be easily modelled as T-dimensional vectors – where T is the total amount of available samples –, and the t^{th} component of the vector represents the sample at time t; free text can be represented as W-dimensional vectors – where W is the total amount of words/bigram/trigram/... in the text –, and the w^{th} component of the vector represents the frequency of the w^{th} word in the text (according to some ordering of words in the text); graphs can be described by $N \times N$ adjacency matrices—where N is the total amount of nodes into the graph. These data representation approaches are inherently distributed and non-rigid. In fact, for any two different time series (possibly sampling the

[5]An n-dimensional vector $\mathbf{x}$ of categorical values $x_1, \ldots, x_n$ where each $x_i \in \mathcal{C} = \{c_1, \ldots, c_m\}$ can always be *one-hot encoded* into a $m \times n$ matrix where the item in position i, j is 1 iff $x_i = c_j$, or 0 otherwise.

similar phenomena), the amount of available samples may be different. Similarly, two different text may involve different sets of words, resulting in vectors of different sizes. Finally, two different graphs (possibly describing similar situations), may involve a different amount of nodes.

Depending on the nature of the data itself, and on the particular data-analytic goal data representation is serving, datasets of such sorts can be translated into rigid form by following one of the strategies below:

draw a number of statistics on each datum (e.g. mean, standard deviation, min, max, etc.), attempting to aggregate the information therein contained: if the same amount and sorts of statistics are drawn for each datum, the dataset will then become tabular;

apply a domain-chancing transformation such as the Fourier transform [CLW69] or the wavelet transform [Zha19];

sub-sample each variable-size datum using a fixed-size sampling step; e.g. a sliding window for time series [FDH01], or neighbourhoods of fixed sizes for graphs;

exploit *ad-hoc* embeddings targetting particular sorts of data, such as GNN [WPC$^+$21], Word2Vec [Chu17], etc.

3.3 Comparison: Symbolic vs. Sub-Symbolic KR

Symbolic and sub-symbolic approaches to KR can be compared along several dimensions, along which their duality seems clear.

Crispness vs. Fuzziness. At the syntactical level we describe symbolic KR as “flexible” – mostly because it can represent knowledge intensionally, via variables, and concisely, via recursive structures – and sub-symbol KR as “rigid”—because of its prominent reliance on fixed-size arrays. However, it is well understood how, in practice, symbolic KR leads to *crisp* representations, whereas sub-symbolic KR leads to *fuzzy* representations. The distinction is well-established within the AI literature. For instance, in the early 90s, Minsky described symbolic approaches as neat, and sub-symbolic ones as scruffy [Min91].

Regardless of the particular terminological choices, the statement stems from the exact nature of symbolic KR as opposed to the approximate nature of its sub-symbolic counterpart. Indeed, while the interpretation of logic formulæ is often discrete and finite-valued – and more commonly Boolean (i.e. two-valued) –, arrays of real numbers may span through an infinity of values, and continuos notions of similarity or distance may be defined among them. So, for instance, while symbolically represented objects can only be either equals or not, vectors, matrices, or tensors may be more or less similar, according to a continuum of possibilities.

Accordingly, logic-based representation are most adequate to represent exact situations, where the world can be modelled according to precise rules. Conversely, array-based representation are most adequate to represent approximate situations, where similarity or slight differences among entities are interesting and should be explicitly captured.

Extensional vs. Intensional. Another relevant distinction concerns the capability of representing knowledge intensionally. While symbolic approaches support both intensional and extensional representations, sub-symbolic approaches only support extensional representations. This implies that, when represented sub-symbolically, all data should be represented explicitly.

The explicit representation of *all* the available information is at the same time a blessing and a curse. In fact, while it costs far more space – thus complicating both storage and processing –, it simplifies the design of sub-symbolic algorithms, which can rely on the assumption that all relevant data is immediately available.

About Conversions. Conversions among the symbolic and sub-symbol realm (or vice versa) are where discrepancies become more evident. In particular, while the conversion in symbolic form of some sub-symbolic array of number is always possible – as extensive tabular information can be suitably represented via logic formulæ as well –, the conversion of logic knowledge into sub-symbolic form is cumbersome. Despite many conversion strategies (or embeddings) have been proposed into the literature, they commonly require:

1. all the constants, functor symbols, and predicate symbols to be explicitly encoded [CDM20, sec. 6.2],
2. variables to be missing, as they would imply some intensional representation [SdG16],
3. N-ary structures to be encoded into tensors having at least N dimensions, possibly recursively combined via the tensor product [Smo90].

Unfortunately, all such requirements come with quite strong limitations. In particular, item 1 implies that constants, functors, and predicate symbols must be of finite quantity and all *a-priori* known—both conditions which rarely hold in practice. Item 2 implies logic formulæ should be *grounded*, if not already ground—which would obliterate the advantages coming from intensional representations.

Finally, item 3 implies that the maximum level of recursion should be *a-priori* defined, as each tensor product increases the dimensionality of the tensors at hand—which in turn cannot increase indefinitely, as it would violate the rigidity required by sub-symbolic KR. All such issues arise because sub-symbolic KR is inherently extensional, and it involves no simple way to express data intensionally – and therefore concisely – as in logic.

Chapter 4

Learning Knowledge from Data

A famous definition of machine learning from [Mit97] states:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E

This definition is very wide, as it does not specify *(i)* what are the possible tasks, *(ii)* how performance measured is in practice, *(iii)* how / when experience should be provided to tasks, *(iv)* how exactly the program is supposed learn, and *(v)* under which form learnt information is represented. Accordingly, depending on the particular ways these aspects are tackled, a categorisation of the approaches and techniques for letting software agents learn may be drawn.

As depicted in fig. 4.1, three major approaches to ML exist. Each approach is characterised by a well-defined pool of tasks, which may, in turn, be applied in wide range of use case scenarios. The three major approaches to learning are: *supervised*, *unsupervised*, and *reinforcement*. They essentially deal with the kind of task T to be learned – commonly consisting in the estimation of an unknown relation –, and how experience E is provided to the learning algorithm.

In supervised learning, the learning task consists of finding a way to approximate an unknown relation, given a sampling of its items—which constitute the experience. In unsupervised learning, the learning task consists of finding the best relation for a sample of items – which constitute the experience –, following a given optimality criterion intensionally describing the target relation. In reinforcement learning, the learning task consists of letting an agent estimate optimal plans given the reward it receives whenever it reaches particular goals—constituting the experience. There, a plan can be described as a relation among the possible states of the world, the actions to be performed in those states, and the reward the agents expects to receive from that action.

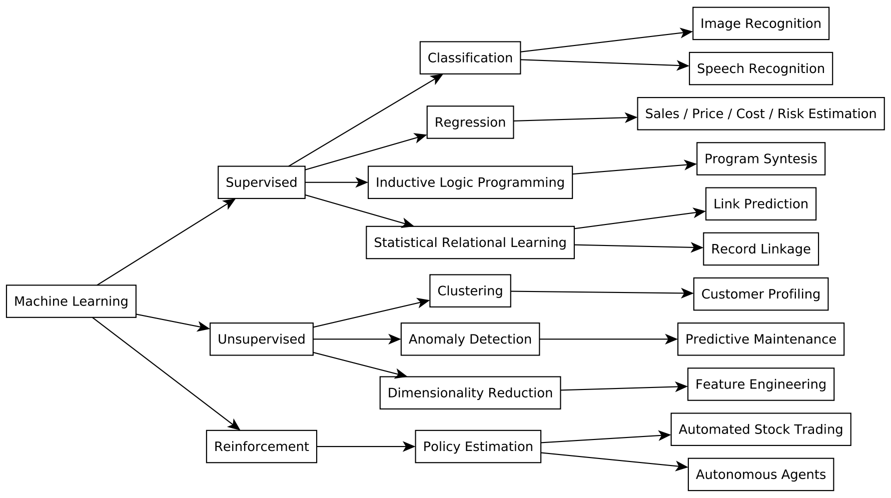


Figure 4.1: Taxonomy of ML. The second column enumerates the three major families of ML approaches, the third one enumerates the main sorts of tasks affiliated with each family, whereas the fourth one enumerates possible applications for each task

Such categorisation of learning approaches can be applied to both symbolic and sub-symbolic techniques. Indeed, in this chapter, we provide an overview of learning on a per-representation basis. In particular, in the following sections we summarise the state of the art for what concerns both symbolic and sub-symbolic forms of *supervised* learning.

4.1 Sub-Symbolic Supervised Machine Learning

Since several practical AI problems – such as image recognition, financial and medical decision support systems – can be reduced to *supervised* ML – which can be further grouped in terms of either *classification* or *regression* problems [Twa10, Kot07] –, in the reminder of this section we focus on this set of ML problems.

Within the scope of sub-symbolic supervised ML, a *learning algorithm* is commonly exploited to approximate the specific nature and shape of an unknown *prediction* function (or *predictor*) $\pi^* : \mathcal{X} \to \mathcal{Y}$, mapping data from an input space $\mathcal{X}$ into an output space $\mathcal{Y}$. There, common choices for both $\mathcal{X}$ and $\mathcal{Y}$ are, for instance, the set of vectors, matrices, or tensors of numbers of a given size—hence the sub-symbolic nature of the approach.

An important assumption significantly affecting both the theory and the practice of sub-symbolic supervised learning is that vectors, matrices, or tensors in $\mathcal{X}$ and $\mathcal{Y}$ are of *fixed* size—despite items in $\mathcal{X}$ may have different sizes than the items in $\mathcal{Y}$. Without lack of generality, in what follows we refer to items in $\mathcal{X}$ as n-dimensional vectors denoted as $\mathbf{x}$, whereas items in $\mathcal{Y}$ are m-dimensional vectors denoted as $\mathbf{y}$—despite matrices or tensors may be suitable choices as well.

To approximate function π^*, supervised learning assumes a learning *algorithm* is in place. This algorithm computes the approximation by taking into account a number N of *examples* of the form $(\mathbf{x}_i, \mathbf{y}_i)$ such that $\mathbf{x}_i \in X \subset \mathcal{X}$, $\mathbf{y}_i \in Y \subset \mathcal{Y}$, and $|X| \equiv |Y| \equiv N$. There, the set $D = \{(\mathbf{x}_i, \mathbf{y}_i) \mid \mathbf{x}_i \in X, \mathbf{y}_i \in Y\}$ is called *training* set, and it consists of $(n+m)$-dimensional vectors. The dataset can be considered as the concatenation of two matrices, namely the $N \times n$ matrix of *input* data (X) and the $N \times m$ matrix of *expected output* data (Y). There, each $\mathbf{x}_i$ represents an instance of the input data for which the expected output value $\mathbf{y}_i \equiv \pi^*(\mathbf{x}_i)$ is known or has already been estimated. Notably, such sorts of ML problems are said to be "supervised" *because* the expected outputs Y are available. Furthermore, the function approximation task is called "regression" if the components of Y consist of continuous or numerable – i.e. *infinite* – values, or "classification" problems they consist of categorical – i.e. *finite* – values.

Many learning algorithms exist, and they work in quite different ways. However, the general layout of sub-symbolic supervised learning is the same in all cases. The learning algorithm assumes π^* to be a function from a given set of functions $\mathcal{H}$ called *hypotheses* space—i.e. $\pi^* \in \mathcal{H}$. In other words, the underlying assumption is that the unknown prediction function π^* exists, and it is of the form characterising all functions in $\mathcal{H}$. The algorithm performs an exploration of the *hypotheses* space $\mathcal{H}$ looking for the hypothesis function $\hat{\pi} \in \mathcal{H}$ that better fits the data in D—and that, therefore, better approximates π^*.

The goodness of the fitting among a hypothesis function $\hat{\pi}$ and the data can be assessed via either *(i)* an error function $\varepsilon : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ measuring the discrepancy among the expected outputs in Y and the values attained by applying $\hat{\pi}$ to X, or dually, *(ii)* an adherence function $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\leq 1}$, measuring the similarities among the same values. For the sake of simplicity, we here consider the best hypothesis $\hat{\pi}$ the one item of $\mathcal{H}$ for which the total error is minimal – or the total adherence is maximal –, w.r.t. the data in D. Therefore, in theory, any sub-symbolic supervised learning process can be abstractly described via any of the following formulæ:

$$\hat{\pi} = \operatorname*{argmin}_{\pi \in \mathcal{H}} \left\{ \sum_{i=1}^{N} \varepsilon(\mathbf{y}_i,\ \pi(\mathbf{x}_i)) \right\} \quad \text{or} \quad \hat{\pi} = \operatorname*{argmax}_{\pi \in \mathcal{H}} \left\{ \sum_{i=1}^{N} \rho(\mathbf{y}_i,\ \pi(\mathbf{x}_i)) \right\}$$

Parameters and hyper-parameters. Exploration of the hypothesis space is what is commonly referred to as "learning" or "training". Learning algorithms mostly differ for the strategy they follow to perform such exploration, other than the particular hypotheses spaces they support.

A common strategy followed by most learning algorithms leverages on the assumption that the hypothesis space is the set of all functions having the same shape, regulated by a given amount p of *parameters*—namely $\mathcal{H}_\Theta$ where $\Theta \subseteq \mathbb{R}^p$ is space of parameters, enumerated by θ. Under such assumption, the formulation of supervised learning can be rewritten as the optimisation task of finding the optimal parameters θ^* among the ones in Θ:

$$\theta^* = \operatorname*{argmin}_{\theta \in \Theta} \left\{ \sum_{i=1}^{N} \varepsilon(\mathbf{y}_i,\ \pi_\theta(\mathbf{x}_i)) \right\} \quad \text{or} \quad \theta^* = \operatorname*{argmax}_{\theta \in \Theta} \left\{ \sum_{i=1}^{N} \rho(\mathbf{y}_i,\ \pi_\theta(\mathbf{x}_i)) \right\}$$

where $\pi_\theta \in \mathcal{H}_\Theta$ is the particular function using the parameters in θ.

If all functions in $\mathcal{H}_\Theta$, as well as the error (resp. adherence) function ε (resp. ρ), are differentiable w.r.t. θ, then the optimisation task can be tackled via gradient descent (resp. climbing) in the general case—despite better options may exist for particular shapes of the functions in $\mathcal{H}_\Theta$. Such need to rely on differentiable functions of vectors of real numbers is what forces many ML techniques into the sub-symbolic realm.

Notably, a hypothesis space is commonly generated by a particular assignment of a number q of *hyper-parameters* $\omega \in \mathbb{R}^q$. Each particular value of ω corresponds to a particular parameters space Θ, and therefore to a particular hypothesis space $\mathcal{H}_\Theta^\omega$. The hypothesis space may consist for instance of the set of all polynomials of a variables whose degree is b. That would imply the corresponding parameters space to comprehend all possible vectors having $\binom{b}{a}$ components. So, if $a = b = 1$, then $\mathcal{H}_\Theta^\omega$ is the set of all possible straight lines on a plane, i.e. polynomials parametrised by θ_1 and θ_2. For $a = 1$ and $b = 2$, $\mathcal{H}_\Theta^\omega$ corresponds to the set of all possible parabolas on a plane, i.e. polynomials parametrised by θ_1 and θ_2 and θ_3. A similar example may be built upon polynomials of 2 variables, and so on.

The difference among parameters and *hyper*-parameters is very important in practice. In fact, while parameters are *automatically* computed by the learning algorithm, hyper-parameters are not. They may be either guessed or estimated by trial-and-error by data scientists—hence representing a bottleneck in the automatisation of learning.

4.1.1 Overview on learning algorithms

Depending on the predictor family of choice, the nature of the admissible hypotheses spaces and learning algorithms may vary dramatically, as well as the predictive performance of the target predictor, and the whole efficiency of learning.

In the literature of machine learning, statistical learning, and data mining, a plethora of learning algorithms have been proposed along the years. Because of the "no free lunch" (NFL) theorem [WM97], however, no algorithm is guaranteed to outperform the others in all possible scenarios. For this reason, the literature and the practice of data science keeps leveraging on algorithms and methods whose first proposal was published decades ago. Most notable algorithms include for instance (deep) neural networks, decision trees, (generalised) linear models, nearest neighbours, support vector machines (SVM), random forests, and many others.

These algorithms can be categorised in several ways, for instance depending on *(i)* the supervised learning task they support (classification vs. regression), *(ii)* on when they consume data (lazy vs. eager), *(iii)* or on the underlying strategy adopted for learning (e.g. gradient descent, least squares optimisation), etc.

Some learning algorithms (e.g. neural networks) target regression problems, whereas others (e.g. SVM) target classification problems. Similarly, some target multi-dimensional outputs ($\mathbf{y} \in \mathbb{R}^m$), whereas others target mono-dimensional outputs ($m = 1$). Regressors are considered as the most general case, as other learning tasks can usually be defined in terms of mono-dimensional regression. Binary classifiers, for instance, can be treated as mono-dimensional regressors where admissible outputs lay in the interval $[0, 1]$, while multi-class (resp. multi-dimensional) classifiers (resp. regressors) can be treated as ensembles of multiple binary classifiers (resp. regressors).

The eager–lazy dichotomy relates to operational aspects of learning, and, in particular, to *when* training data is actually processed. This, in turn, affects the computational time required in the training and inference phases—i.e. when the predictor is exploited to draw predictions. In principle, the training phase of *lazy* predictors (e.g. nearest neighbours) is trivial and no data needs to be processed as training data is mostly processed in the inference phase. This makes the training phase quicker, at the expense of a slower inference phase. To mitigate this issue, in practice, indexing or grouping of training data may be exploited in the training phase, with the purpose of speeding up the inference phase. Conversely, *eager* predictors (e.g. linear models, neural networks, and virtually any other method mentioned so far) come with a full-fledged learning phase, where the unknown function binding input and outputs is approximated from training data. There, the learning phase carries the higher computational effort, and the inference phase is quick. In the reminder of this section, we focus on eager predictors as they actually produce an internal representation of data during their learning phase, which is the starting point of relevant discussion carried out in chapter 6.

Finally, the learning strategy is inherently bound to the predictor family of choice. Neural networks, for instance, are trained via back-propagation [RHW86] – a particular case of stochastic gradient descent (SDG, [Wik21c]), tailored on NN –, generalised linear models via Gauss' least squares method, decision trees via CART [BFOS84], etc. Despite all such algorithms may appear interchangeable in principle

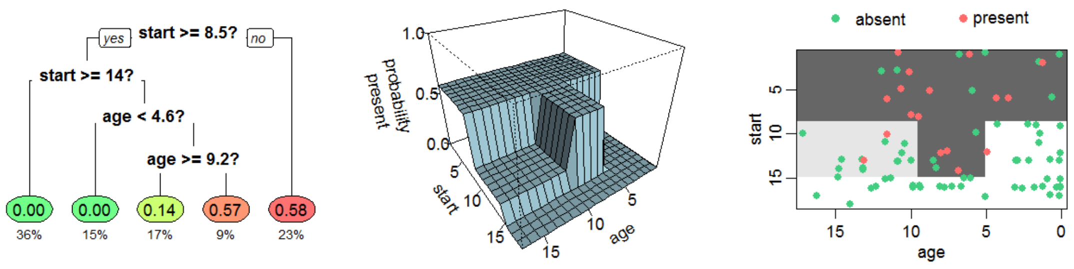


Figure 4.2: An example decision tree estimating the probability of kyphosis after spinal surgery, given the *age* of the patient and the vertebra at which surgery was *start*ed [Wik21b]. Notice that each decision tree subtends a partition of the input space, and that the tree itself provides an intelligible representation of *how* predictions are attained

– because of the NFL theorem –, their malleability is very different in practice. For instance, the least squares method involves inverting matrices of order N – where N is the amount of available examples in the training set –, making the computational complexity of learning more than quadratic in time. Furthermore, in practice, convergence of the method is not guaranteed in the general case, while it is for generalised linear models—hence why it is not adopted elsewhere. Thus, learning by least squares optimisation may become impractical for big datasets or for predictor families outside the scope of generalised linear models. Conversely, the SGD method involves arbitrarily-sized subsets of the dataset (a.k.a. batches) to be processed a limited (i.e. controllable) amount of times. Hence, the complexity of SGD can be finely controlled and adapted to the computational resources at hand—e.g. by making the learning process incremental, and by avoiding all data to be loaded in memory. Furthermore, SGD can be applied to several sorts of predictor families (there including neural networks and generalised linear models), as it only requires the target function to be differentiable w.r.t. its parameters. For all these reasons, despite the lack of optimality guarantees, SGD is considered as very effective, scalable, and malleable in practice, hence why it is extensively exploited in the modern data science applications.

In the reminder of this thesis, we focus on two particular families of predictors – namely, decision trees and neural networks –, and their respective learning methods—i.e. the CART [BFOS84] and back-propagation [RHW86] algorithms. Notably, decision tree are relevant because of their user friendliness, whereas neural networks are relevant because of their predictive performance and flexibility.

Decision Trees. Decision trees (DT) are particular sorts of predictors supporting both classification and regression tasks. In their learning phase, the input space is recursively *partitioned* through a number of splits (a.k.a. *decisions*) based on

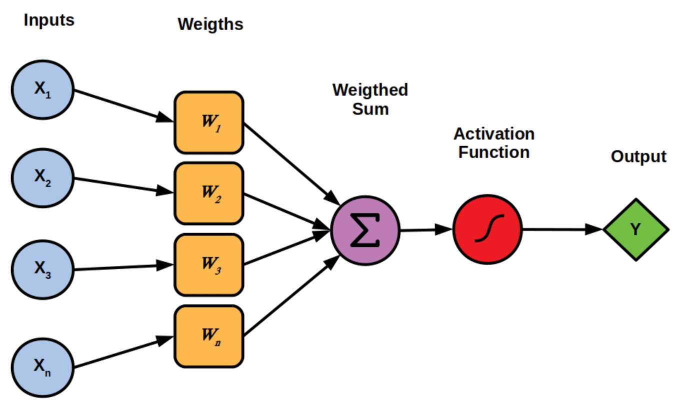


Figure 4.3: General structure of neural units in neural networks

the input data X, in such a way that the prediction in each partition is constant, and the error w.r.t. the expected outputs Y is minimal, while keeping the total amount of partitions low as well. The whole procedure then synthesises a number of *hierarchical* decision rules to be followed whenever the prediction corresponding to any $x \in \mathcal{X}$ must be computed. In the inference phase, decision rules are orderly evaluated from the root to some leaf, in order to select the portion of the input space $\mathcal{X}$ containing x. As each leaf corresponds to a single portion of the input space, the whole procedure results in a single prediction for each x.

Differently from other families of predictors, the peculiarity of decision trees lays in the particular outcome of the learning process – namely, the *tree* of decision rules – which is naturally intelligible for humans and graphically representable in 2D charts. As further discussed in the reminder of this thesis, this property is of paramount importance whenever the inner operation of an automatic predictor must be interpreted by a human being.

Neural Networks. Neural networks (NN) are biologically-inspired computational models, made of several elementary units (neurons) interconnected into a graph (commonly, *directed* and *acyclic*, a.k.a. DAG) via *weighted* synapses. Accordingly, the most relevant aspects of NN concern the inner functioning of neurons and the particular architecture of their interconnection.

Neurons are very simple numeric computational units. They accept n scalar inputs $(x_1, \ldots, x_n) = \mathbf{x} \in \mathbb{R}^n$ weighted by as many scalar weights $(w_1, \ldots, w_n) = \mathbf{w} \in \mathbb{R}^n$, and they process their linear combination $\mathbf{x} \cdot \mathbf{w}$ via an activation function [Wik21a] $\sigma : \mathbb{R} \mapsto \mathbb{R}$, producing a scalar output $y = \sigma(\mathbf{x} \cdot \mathbf{w})$, as depicted in fig. 4.3. The output of a neuron may become the input of many others, possibly forming *networks* of neurons having arbitrary topologies. These network may be fed with any numeric information encoded as vectors of real numbers by simply letting a number of neurons produce constant outputs.

While virtually all topologies are admissible for NN, not all are convenient. A number of convenient *architectures* – roughly, patterns of well-studied topologies – have been proposed into the literature [VVL19] to serve disparate purposes—far beyond the scope of supervised machine learning. Figure 4.4 overviews the current state of the art of NN architectures.

NN can be *trained* on numeric data via stochastic gradient descent and exploited into both supervised and unsupervised learning tasks such as classification, regression, and anomaly detection, depending on the particular architecture of choice. More precisely, while the training *automatically* sets up the weights of each neuron's ingoing synapses, the overall topology of the network is not allowed to vary. It is rather assumed to be *manually* engineered by data scientists.

Most common NN architectures are feed-forward, meaning that neurons are organised in successive *layers*, in such a way that neurons from layer i can only accept ingoing synapses from neurons of layers $j < i$. The first layer is considered the input layer, which is used to *feed* the whole network, while the last one is the output layer, where prediction are drawn. In architectures of these kinds, inference lets information flow from the input to the output layers – assuming the weights of synapses are fixed –, while training lets information flow from the output to the input layers—provoking the variation of weights to minimise the prediction error of the overall network.

The recent success of deep learning [GBC16] has proved the flexibility and the predictive performance of *deep* neural networks (DNN). 'Deep' here refers to the large amount of (possibly *convolutional*) layers. In other words, DNN can learn how to apply cascades of convolutional operations to the input data. Convolutions let the network spot relevant features into the input data, at possibly different scales. Hence why DNN are good at solving complex pattern-recognition tasks, such as in computer vision or speech recognition. Unfortunately, however, unprecedented predictive performances of DNN come at the cost of their increased internal complexity and greater data greediness.

4.2 Symbolic Supervised Learning

Within the realm of symbolic AI, supervised ML commonly refers to either *inductive logic programming* (ILP) [Mug91] or *statistical relational learning* (SRL) [DRK10], despite the overlap among the two disciplines is wide.

In both cases, learning consists of approximating an unknown *intensional* representation H^* for a number of positive examples E^+, possibly leveraging on

- some prior knowledge base B, carrying the so-called *background knowledge* about the domain at hand;

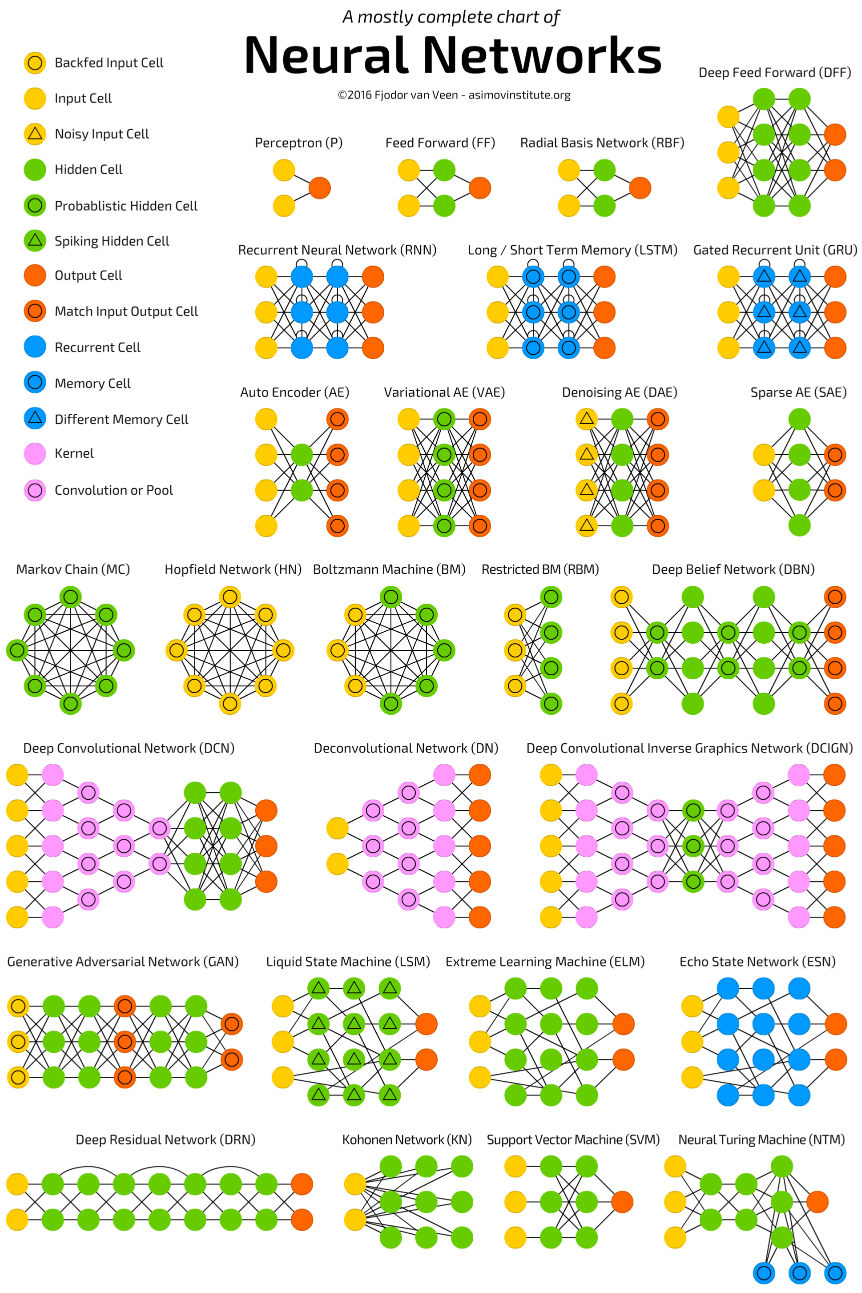

A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Figure 4.4: Admissible architectures for neural networks [VVL19]

- a number of *negative* examples E^-;
- a *language bias* C, constraining the admissible shapes for the representation to-be-learned.

In the general case, H^*, E^+, E^-, and B are knowledge bases, possibly involving several definite clauses (i.e. either rules or facts), while the shape of C really depends on the particular learning approach at hand. Of course, there may be cases where E^-, B or C are not required, and therefore considered as empty sets.

The main difference among ILP and SLR lays in the particular language used for knowledge representation. While in ILP knowledge bases simply consist of bare definite clauses, SRL leverages on a superset of Horn Logic called LPAD (Logic Programs with Annotated Disjunctions) [VVB04], where definite clauses are enriched with probability values.

According to the SLR nomenclature of [DRK10] – where a unifying model generalising ILP and SLR is proposed –, there are two relevant problems which lay under the symbolic supervised learning umbrella, namely:

parameters learning where H consists of a given LPAD theory, where the shape of facts and rules is known, while their probabilities are not, and learning aims at simply estimating those probabilities

structure learning where H is completely unknown, and the whole shape of facts and rules therein contained must be computed, possibly along with their corresponding probabilities

In both cases, learning can be defined as an optimisation problem aimed at approximating H^* by search for the best KB $\hat{H}$ into a hypothesis space $\mathcal{H}_{B,C}$, defined by applying all possible combinations of clauses in B, as dictated by C. There, each KB $H \in \mathcal{H}_{B,C}$ consists of a number of definite clauses defining the n-ary relation h, possibly leveraging on the relations defined in B, and satisfying the suggestions/constraints expressed by C. In other words, they consist of KB intensionally defining h via rules of the form:

$$\psi :: h(X_1,\ \ldots,\ X_n) \leftarrow f(\bar{X}),\ f'(\bar{X}'),\ f''(\bar{X}''),\ \ldots$$

where ψ denotes an optional probability value, while $\bar{X}, \bar{X}', \bar{X}'', \ldots$ are tuples involving one or more head variables (i.e., $X_1, \ldots, X_n$), and $f, f', f'', \ldots$ are either predicates defined in B or combinations of those predicates, attained by following the suggestions/constraints contained in C. Similarly, E^+ (resp. E^-) consists of facts of the form h/n, extensionally defining known (resp. invalid) items of the n-aray relation h, and possibly labelled with probabilities.

Analogously to the sub-symbolic case, symbolic supervised learning leverages upon some adherence function ρ, aimed at measuring the adherence of some hypothesis KB $H \in \mathcal{H}_{B,C}$ w.r.t. either E^+ or E^-. For instance, in SRL, ρ is commonly modelled probabilistically:

$$\rho(H, E, B) = \sum_{e \in E} \mathbb{P}(e \mid H, B)$$

where $\mathbb{P}(\cdot \mid \cdot)$ denotes the conditional probability operator. Conversely, in ILP, ρ is modelled in terms of logic inference[1]:

$$\rho(H, E, B) = \sum_{e \in E} \rho(H, e, B) \quad \text{and} \quad \rho(H, e, B) = \begin{cases} 1 & \text{if } H, B \models e \\ 0 & \text{otherwise} \end{cases}$$

Under such hypothesis, symbolic supervised learning can be defined as [DRK10] the optimisation task aimed at finding the hypothesis which adheres to as much positive examples as possible, while adhering to no negative example at all:

$$\hat{H} = \underset{H \in \mathcal{H}_{B,C} \text{ s.t. } \rho(H, E^-, B) = 0}{\operatorname{argmax}} \left\{ \rho(H, E^+, B) \right\} \tag{4.1}$$

Parameter and structure learning differ for the actual way the search is performed, other than for the actual object of search. In parameter learning, algorithms can assume the shape of (facts and rules in) $\hat{H}$ to be given (and fixed), and therefore focus on the mere estimation of probabilities. Conversely, in structure learning, algorithms must also consider the many possible shapes $\hat{H}$ may have. This includes all possible combinations of all relations possibly defined in B. Assuming, for instance, that B intensionally defines r relations $f_1, \ldots, f_r$, whose arity is at least a, and that the target relation is h, whose arity is n. Under such hypothesis, rules in H should be of the form:

$$h(X_1, \ \ldots, \ X_n) \leftarrow \ldots$$

where the body of the rule may contain as many predicates as in any possible permutation of any possible subset of $\{f_1, \ldots, f_r\}$. There, each possible a-ary relation could be written as a predicate involving some disposition of a variables from the set $\{X_1, \ldots, X_n\}$. In other words, the search space for structure learning is *huge* and definitely impossible to explore in useful time, unless in trivial cases. To complicate the matter, differently from the sub-symbolic case, the search space is

[1]This aspect is better discussed in chapter 5. Within the scope of this chapter, the notation $K \models \phi$, where K is a knowledge base and ϕ is a logic formula, can simply be read as "ϕ can be inferred from K via some inference procedure".

not even continuous—meaning that gradient based approaches cannot be applied.

To mitigate such issues – and to reduce the search space –, the ILP community leverages on smart choices of the linguistic bias C. The general purpose of the linguistic bias is to constrain the particular way relations from B can be combined in H. This can come in different shapes and flavours, depending on the particular ILP method in place.

Consider for instance the case of an ILP problem aimed at learning the positive example $grandparent(\mathtt{abraham}, \mathtt{jacob})$, given the background knowledge containing a number of facts expressing parenthood facts of the form $parent(\mathtt{p}, \mathtt{p'})$, describing Abraham's family tree—as in eq. (3.3). There, the meta-rule 4.2 may suggest the correct identification of the target rule – namely, $grandparent(X, Y) \leftarrow parent(X, Z), parent(Z, Y)$ – via the variable assignment $\{\mathsf{P} \mapsto grandparent, \mathsf{Q} \mapsto parent, \mathsf{R} \mapsto parent\}$.

It is worth to be noted how the language bias C plays, in symbolic supervised learning, the same role played by hyper-parameters in sub-symbolic supervised learning. In both cases, *automated* learning relies on some prior knowledge, which must be handcrafted by data scientists. In fact, similarly to sub-symbolic approaches, some mechanism is needed to let humans control either the complexity or learning or the dimension of the search space. In the particular case of symbolic supervised learning, that mechanism is the language bias.

4.2.1 Overview on learning algorithms

Here we summarize the most relevant sorts of algorithms supporting symbolic supervised learning. In particular, we focus on algorithms aimed at learning either the structure or the parameters of logic programs. We discuss approaches targeting structure learning first. Then, we introduce approaches for parameter learning—which commonly require the structure to be given.

Structure Learning

Structure learning of logic programs is commonly attained via some ILP algorithm. A nice and up-to-date survey of ILP is provided by [CD20]. Here, we just provide an overview and some references.

Generally speaking, ILP algorithms aim to construct a good – i.e. *sufficiently* general – theory H^* entailing all positive examples in E^+ and no negative one from E^-. Clauses from the theory under construction may leverage upon one or more clauses from the background knowledge B. In other words, ILP algorithms exploit automated inductive reasoning to produce novel symbolic knowledge – possibly leveraging on previously available symbolic knowledge – out of positive and negative symbolic examples.

Notably, the notion of "*sufficiently* general theory" is quite subtle in this context. Roughly speaking, a *general* theory should contain one or more non-ground clauses, from which the positive examples could be attained via variables substitutions or via deduction. Despite the expected result of any ILP algorithm is certainly a *general* theory from which all the positive examples can be inferred, looking for the *most* – or *least* – possible general rule is likely a mistake. In other words, as for sub-symbolic ML, generalising too much is as wrong as generalising too little—i.e. drawing rules which are too specific. In fact, any inductive inference process may lead to the construction of too general conclusions which do not hold in the real world. To prevent this issue, ILP algorithms commonly include some stopping criterion aimed at avoiding both excessive generalization and excessive generalization.

From a very high-level perspective, ILP algorithms can be categorized w.r.t. the strategy they follow in constructing the target theory. Top-down algorithms start from a very general theory and they progressively specialise it, until reaching the most general theory among the most specific ones. Conversely, bottom-up algorithms follow the inverse path, starting from a very specific theory and then progressively generalising it, until reaching the most general theory among the most specific ones.

In practice, four major approaches have been proposed into the ILP literature – namely, relative least-general generalization, inverse entailment, bottom-clause propositionalisation, and meta-interpretative learning –, and most algorithms proposed so far rely on some of them.

Relative least-general generalization (RLGG). RLGG [Plo71] is a basic mechanism to be exploited in a bottom-up induction strategy. It assumes both the background knowledge and the examples to be ground. Under this assumption, it subtends a lattice where vertices represent clauses and arcs represent instances of a subsumption relation. Such clauses attained by *(i)* combining the literals from the positive examples among each other and with the literals of the clauses in the background knowledge, and by *(ii)* replacing common constant terms with variables, accordingly.

To perform induction, the resulting lattice should then be explored looking for the best RLGG, i.e. the one covering the more positive examples, while not covering any negative example. Unfortunately, however, the lattice may be very large or even infinite, thus further sub-strategies should be in place to *(i) lazily* generate and explore the lattice, *(ii)* prevent it from exploding in size, and *(iii)* prune the lattice as quickly and as much as possible.

Golem [MF92] is an algorithm and former software system exploiting RLGG in practice. In a seek for tractability, Golem puts some constraints on the amount and the position of variables in the literal composition phase. It starts by constructing a very large clause using positive examples, and, after a number of generalization steps where constant terms are replaced by variables, the clause reaches its final form as the outcome of the induction process. Behind the scenes, negative examples are exploited to prune the resulting clause.

Inverse entailment (IE). IE [Mug95] is another basic mechanism supporting the induction of logic clauses following a bottom-up strategy. Like RLGG, IE subtends a generality lattice of clauses to be explored. However, differently than RLGG, IE may generalize a clause via *predicate invention*—i.e. by generating a bare new predicate, different than the ones in the examples and background knowledge. Thanks to predicate invention, IE can lead to the induction of elegant and concise theories composed by one or more interrelated clauses.

While the IE mechanism itself is straightforward, it simply moves the complexity into predicate invention. Indeed, linguistic bias in the form of suggestions and constraints for the predicate invention sub-procedure must be commonly provided by the users.

Progol, for instance, is likely the first and most relevant ILP system leveraging upon IE. It has been proposed by the same paper [Mug95]. Notably, it requires the user to provide "mode declarations" via an ad-hoc syntax, to define and constrain predicate invention. This is required to construct the "bottom clause", i.e. most-specific clause that explains an example. Progol then relies upon an A^*-like heuristic search – exploiting a compression score as the heuristic function – to generalise the clause to make it cover as much examples as possible.

Bottom-clause propositionalisation (BCP). In the BCP approach [FZdG14], bottom clauses attained by generalising the examples are *propositionalised* – i.e. brought into attribute-value form – by using the set of all body literals that occur in them and in the background knowledge's clauses as possible attributes. This reduces the logic knowledge into an extensional dataset of fixed size, which can be used to fed sub-symbolic predictors.

C-IL^{2}P [dGZ99] and CILP++ [FZdG14] are notable examples of systems perform ILP by relying on BCP and by constructing a neural network behind the scenes. In these systems, neural networks are constructed to reflect the background knowledge's and the examples' literals into its structure. The neural network is then trained, and an induced clause is then reverse-engineered from the network's weights.

Meta-interpretative learning (MIL). MIL [MLPT14] is a modern approach leveraging on the meta-programming capabilities of logic solvers (cf. chapter 5)—

and in particular Prolog ones. Following the MIL approach, ILP is conceived as a higher-order logic task where a logic program (the meta-interpreter) dynamically constructs another logic program (the induced theory) via meta-programming.

There, the linguistic bias consists of a library of higher-order rules (a.k.a. meta-rules), which define the admissible ways the predicates from the examples and the background knowledge may be combined. In particular, a meta-rule is a rule involving higher-order variables enumerating over predicate symbols, such as:

$$\mathsf{P}(A,\ B) \leftarrow \mathsf{Q}(A,\ C),\ \mathsf{R}(C,\ B) \tag{4.2}$$

where uppercase, sans-serif letters $\mathsf{P}, \mathsf{Q}, \mathsf{R}$ denote higher-order variables, while A, B, C are ordinary variables; and the whole formula allows an induction algorithm to *invent* [MB88] some binary predicate P by combinations of two binary predicates Q and R.

Metagol [MLT15] is the most prominent example of a system following the MIL approach. It follows a bottom-strategy starting from the positive examples and using meta-rules to guide the generalization process. After generalising all the examples, Metagol checks the consistency of the induced theory against the negative examples. Alternative hypotheses may be explored by the meta-interpreter following the underlying solver's semantics—e.g. backtracking in the case of Prolog.

Parameter Learning

Parameter learning is commonly achieved via probabilistic logic programming (PLP). The history of PLP starts with the seminal work of [NS92], and a nice and up-to-date survey is provided by [Rig18].

In PLP, theories may contain facts or rules enriched with probabilities, which may, in turn, be queried by the users to investigate not only which statements are true or not, but also under which probability. To support this behaviour, probabilistic solvers leverage ad-hoc resolution strategies explicitly taking probabilities into account. PLP systems may also support the computation of probabilities in presence of data. These features make them ideal to deal with uncertainty and the complex phenomena of the physical world. It is thus unsurprising that Bayesian and data-driven AI, other than cyber physical systems (CPS), are among the areas which would benefit the most from the development of robust and interoperable PLP technologies.

A variety of research contributions exploring the field of PLP exist in the logic programming literature. Proposals often differ for their semantics or syntaxes, or for the way they perform probabilistic reasoning [FdBR+15, KDDR+11, DRKT07, RS11].

Roughly speaking, semantics are concerned with endowing probabilistic programs with meaning. Sato's distribution semantics (DS) [Sat95, SK97] is one of the most prominent approaches for the combination of logic programming and probability theory. There, a probabilistic logic program is interpreted as a concise description of many possible worlds, and the probabilities of queries are solved by summing up their probability in each possible world.

Languages adhering to the distribution semantics may in turn differ in how they represent clauses, and their probabilities. A successful approach in this context is LPAD (Logic Programs with Annotated Disjunctions) [VVB04], where clauses admit disjunctions of atoms in their heads, and each atom is labelled with a probability value. In other words, LPAD is a special notation supporting the definition of non-binary probabilistic distributions over clauses and facts. However, in practice, probabilistic logic programs may support a certain *evidence* [Hor16] to be provided via unannotated fact/rules which are known to be true, even though they may be defined over some probability distribution.

Finally, concerning probabilistic reasoning, PLP generally supports reasoning tasks (cf. chapter 5), and each of them has been richly documented in the literature [dK15]. Broadly speaking, options range from exact to approximate—the former being more precise and computationally demanding, while the latter being more affordable at the price of lower precision. In any case, a common strategy is to rely upon knowledge compilation [DM02] to make probabilistic reasoning efficient—i.e., by transforming logic formulæ into simpler (more tractable) forms. Binary decision diagrams (BDD) [Ake78, LMS14] and their variants/extensions are commonly exploited to serve this purpose [BR13, VRVdBDR14].

4.3 Symbolic vs. Sub-Symbolic Learning

Symbolic and sub-symbolic approaches to supervised learning share similar formulations, despite the corresponding methods and algorithms operate in quite different ways. Both formulations deal with optimisation problems aimed at iteratively constructing an algorithm mimicking an unknown relation/function in the best possible way, leveraging on a number of examples. However, because of their nature and the inherent way they represent knowledge, both approaches come with pros and cons. Here we focus, on their flexibility, maturity, data and computational efficiency, degree of automation, and validation.

Flexibility and maturity. For what concerns flexibility, symbolic approaches produce more flexible outcomes, whereas sub-symbolic approaches are characterised by more flexible learning processes.

Outcomes. Focussing on symbolic approaches, flexibility lays in the shape of the expected outcomes, which is a direct effect of the particular choice of symbols for KR. "Symbolic" here implies that knowledge is represented via logic clauses, which in turn pave the way towards learning intensional *relations*—possibly taking some prior (background) knowledge into account. Indeed, representing the target of knowledge in the form of relations expressed by logic formulæ comes with two major advantages—namely bi-directionality and re-usability.

First, relations are *bi-directional*, meaning that any argument of the relation can be considered either an input or an output, depending on the situation at hand. So, for instance, if an agent is capable of learning the clauses expressing the $grandparent/2$ relation – cf. eq. (3.3) –, then it acquires a lot of relevant information, namely: *(i)* an explicit, generic representation of *how* the relation can be tested among any two entities X and Y, *(ii)* a way to compute all the grand-children of any given grand-parent `p` – i.e. $grandparent(\texttt{p}, Y)$ –, and *(iii)* a way to compute all the grand-parents of any given grand-child `c`—i.e. $grandparent(X, \texttt{c})$.

Second, relations are *re-usable* (w.r.t. a learning process), meaning that learned relations can be used as prior knowledge in any sub-sequent learning process, as both the inputs and outputs of any symbolic learning process are represented in the same form—namely, logic clauses. This in turn paves the way towards the definition of learning *cycles* where a learning algorithm is executed several times and the knowledge acquired after each round is included in the background knowledge of successive rounds.

Conversely, sub-symbolic approaches aim to learn *functions*, rather than relations. The learned functions are generally *mono-directional* – in the sense that they are not (easily) invertible – and extensional. Consider for instance the case of a neural network aimed at classifying images of animals. It may easily discriminate among dogs and cats (i.e. compute the classification, given an input), yet it may hardly generate admissible images of neither dogs or cats (i.e. compute an input, given a class)[2].

Processes. Focussing on sub-symbolic approaches, flexibility lays in the variety of methods to approximate the target function. Such variety is once again the result of the particular choice of arrays of numbers for KR. In fact, this choice enables the pervasive exploitation of mathematical operations as the basic bricks of sub-symbolic processing. These include basic algebraic operators (sum, product, etc.), as well as statistical (mean, variance, standard deviation, etc.), information-theoretical (cross-entropy, mutual information, etc.), signal-processing (Fourier- or

[2]Generative Adversarial Neural Networks [GPM+14] may be used whenever bi-directionality is needed, yet that essentially implies training two networks: one *classifier* and one *generator* of data, where the former can only classify, and the latter can only generate data

Laplace-transform, etc.), binary (bitwise-and, -or, etc.), or differential (differentiation, integration, etc.) operators. All such operators, in turn, come with two major advantages in terms of *malleability* and *parallelisation*.

Malleability refers to the capability of learning in spite of how the many elementary operators are combined. Within the scope of sub-symbolic approaches, malleability is commonly guaranteed by the pervasive exploitation of *differentiable* operators, which supports learning via numeric optimisation algorithms—e.g. stochastic gradient descent (SGD) and its variants [Wik21c]. Conversely, within the scope of symbolic approaches, the presence (resp. lack) of any given operator may greatly affect the *expressiveness* of the underlying logic, therefore making learning more (resp. less) complex from a computational perspective.

Parallelisation refers to the capability of speeding up learning algorithms by executing as much sub-tasks as possible in parallel, provided that the adequate hardware is in place. Within the scope of sub-symbolic approaches, most basic mathematical operators – such as matrix- or tensor-products –, as well as whole learning steps – such as *batches* in SGD –, can be executed in parallel to some extent, possible via ad-hoc hardware facilities Conversely, within the scope of symbolic learning, further research on concurrent / parallel solutions is still needed.

Consider, for instance, neural networks as opposed to ILP. They are characterised by a great flexibility because of their reliance on differentiable operators (mostly sums, multiplications and activation functions [Wik21a]), and malleable way of combining them into arbitrarily complex structures. Therefore, regardless of the complexity of the overall structure, a NN is composed by the recursive composition of differentiable operators—which makes the whole network trainable via SGD. To further speed up NN training, a plethora of software frameworks have been designed and implemented, with the purpose of exploiting ad-hoc hardware, such as GPUs—cf. Tensorflow [AAB+15], Theano [ARAA+16], Caffe [JSD+14], etc. Conversely, despite the many algorithms designed for ILP, the availability of software frameworks reifying them is quite scarce, and the support for parallelisation is even scarcer. Should we speculate on the motivations behind this situation, we would argue that symbolic and sub-symbolic approaches to learning have so far reached different levels of *maturity*—especially, for what concerns technological readiness.

Efficiency. Efficiency in learning can be measured against two major aspects, namely time and space. Data (resp. computational) efficiency deals with space (resp. time), and it is related to the amount of data (resp. time) required by learning to be effective. Here, effectiveness refers to the adherence of the learned relation/function w.r.t. the available examples.

Concerning data efficiency, sub-symbolic approaches are notably data-hungry

[Ada21], as they require tons of examples to learn tasks for which a human would require just a handful. Conversely, symbolic approaches are considered far more data-efficient. In [EG18], the authors discuss this notable difference, arguing that a motivation may lay in the strong language bias imposed by choice of logic formulæ as the preferred means for KR.

Concerning computational efficiency, while in theory both symbolic and sub-symbolic approaches must explore *infinite* search spaces, in practice, efficiency can be improved by *(i)* sacrificing effectiveness, e.g. by leveraging on greedy algorithms, strong biases, or aggressive stopping criteria, *(ii)* parallelising the learning algorithm, as discussed above.

Automation and autonomy. A common trait shared by both symbolic and sub-symbolic approaches to supervised learning is their reliance on semi-automatic workflows. In other words, despite the name, both approaches require a "human in the loop" – namely, the data scientist – to take care of those aspects which learning algorithms cannot autonomously deal with. In the case of sub-symbolic approaches, these aspects involve the choice of hyper-parameters. In the case of symbolic approaches, these aspects involve the choice of the language bias and background knowledge. In both cases, these aspects involve the choice of the most adequate learning algorithm(s), other than the engineering of the representation of available data, in maximise the effectiveness the learning algorithm(s).

Within the scope of sub-symbolic learning, the problem of hyper-parameters tuning is currently addressed via a number of practices aimed at automating and speeding up their selection. A summary of such practices can be found in [CM15]. The recent advances in the field of *Automated ML* [HZC21] are building on such practices in order to further increase the degree of automation in sub-symbolic ML. However, current efforts are focussing on supporting data-scientists in an end-to-end fashion, rather than letting software agents learn autonomously.

To the best of our knowledge, automating the definition of background knowledges and language biases in symbolic learning is not a major concern. Should we speculate on the reasons behind this phenomenon, we would argue that both background knowledges and language biases are the preferred way to let human beings transfer their commonsense and wisdom to the learning algorithms. In this sense, the creation of background knowledges and language biases is inherently poorly automatable. However, background knowledges can be incrementally constructed by an agent (be it human or software) and then shared or transferred to other agents. A similar argument may hold for language bias, since it may be considers as meta-level background knowledge—i.e. knowledge about how further knowledge may be constructed.

Chapter 5

Reasoning over Knowledge

The Cambridge dictionary[1] defines *reasoning* as "the process of thinking about something in order to make a decision". Conversely, the Oxford dictionary[2] states that reasoning is "the process of thinking about things in a logical way". Notably, while both definitions agree that reasoning essentially consists in the *process of thinking*, none of them actually constrains the nature of the entity enacting this process, despite thinking – and, in particular, reasoning – is the most characterising capability of *humans*' mind. This welcomes the idea that software agents may be capable of *automated* reasoning as well.

Indeed, within the scope of this thesis, we consider reasoning as the activity performed by an agent (either human, or computational) whenever it draws new knowledge out of prior knowledge. Of course, as for learning, the particular way knowledge is drawn heavily depends on how it is represented.

When knowledge is symbolically represented, reasoning leverages on one or more *inference rules*, i.e. logic formulæ dictating under which conditions conclusions may be drawn out of premises. This reflects the Oxford definition, where the *logic* nature of reasoning is stressed. Inference rules may be used, for instance, to deduce a particular case from a general rule, to induce the general rule justifying a number of observations, or to speculate on the possible causes for some phenomena, given the particular rules governing the underlying noumena. Therefore, within the symbolic realm, the terms "inference" and "reasoning" are used almost interchangeably. Research in this field aims at letting computational agents draw logic inferences automatically. For this reason, the focus of computer scientists is on finding effective and efficient algorithms to let agents reason autonomously—i.e. with minimal human intervention.

Conversely, when knowledge is sub-symbolically represented, "inference" and

[1]`https://dictionary.cambridge.org/dictionary/english/reasoning`

[2]`https://www.oxfordlearnersdictionaries.com/definition/english/reasoning?q=reasoning`

"reasoning" are *reduced* to the data-analytic activity of applying the models learned from previous data to novel data, in order to mine useful information. Such activity is far from trivial, as it may used to perform tasks which would be prohibitively complex to express otherwise—e.g. image recognition. Accordingly, the Cambridge definition is more adequate in this case: reasoning is not necessarily logic in nature, but for sure it is aimed at driving decisions—e.g. deciding whether handwritten characters is more likely a 1 or a 7, or whether an histological image should or should not raise an alarm for cancer. Research in this field aims at letting sub-symbolic algorithms attain better predictive performances. As such, data scientists' efforts are mostly devoted to the improvement of *learning* algorithms, as inference is straightforward. However, recent research efforts are being devoted to the exploitation of sub-symbolic algorithms as means for performing symbolic computations—therefore mimicking logic reasoning.

In the reminder of this chapter we delve into the details of reasoning from both a symbolic and sub-symbolic perspective. In particular, we present the classic approaches and algorithms to automate logical reasoning and we introduce the theory behind the mimicking of symbolic reasoning via sub-symbolic facilities.

5.1 Symbolic Reasoning

This section contains contributions from the following works of ours: [CCDO20]

Symbolic (i.e., logic-based) reasoning approaches root back to John McCarthy's work of 1958 [MS58], aimed at developing the idea of formalising the so-called *commonsense reasoning* to build intelligent artefacts—i.e. computational or cyber-physical agents endowed with human-like intelligence. There, commonsense intuitively refers to the basic understanding of the physical world, its cause-effect rules, and the effects of one's actions on it, etc. [McC89]. It is such an obvious capability for human beings that most of it is not even explicitly taught in schools, yet it is incredibly hard to formalise and represent for computational agents, which are therefore inherently lacking such kind of basic knowledge.

Despite the formalisation of commonsense soon proved to be very challenging – mostly because of the many non-trivial involved issues, such as the need of formalising the situation the agent is immersed into, actions it may perform of be subject to, and physical and legal laws governing its environment and context, etc. –, many frameworks and tools have been developed over the years while pursuing such goal. There are freely available commonsense knowledge bases and natural language processing toolkits, supporting practical textual-reasoning tasks on real-world documents including analogy-making, and other context oriented inferences—see for instance [LLSB04, LS04, LLS02, Sha00]. There have been also a number of attempts to construct very large knowledge bases of commonsense knowledge by hand, one of the largest being the CYC program by Douglas Lenat at CyCorp [Len95].

The modern approach to automated reasoning starts with Robinson's resolution principle [Rob65]: since then, several technologies have exploited *deduction* on FOL knowledge bases to provide reasoning capabilities in diverse areas—logic programming, deductive data bases, and constraint logic programming (CLP) possibly being the major ones. Other approaches and techniques, however, built upon the *induction* and *abduction* principles.

As its name suggests, *deduction* operates top-down, deriving a true conclusion from a universal true premise: logically speaking, this means that the conclusion's truth necessarily follows from the premise's truth. *Induction*, instead, operates bottom-up, basically making a guess – a generalization – from specific known facts: so, the reasoning involves an element of probability, as the conclusion is not based on universal premises. *Abduction* is somehow similar, but seeks for cause-effect relationships—i.e., the goal is to find out under which hypotheses (or premises) a certain goal is provable. Such technologies are exploited, in particular, for the verification of compliance of specific properties [MTC$^+$10].

Logic programming (LP) is likely the most widely-adopted paradigm based on deduction. From Colmerauer and Kowalsky's seminal work [Kow74, Col86], the Prolog language has been since then one of the most exploited language in AI applications [DRW96]. Other valuable approaches include *fuzzy logic*, *answer-set programming* (ASP), *constraint logic programming* (CLP), *non-monotonic reasoning*, and *belief-desire-intention* (BDI).

Fuzzy logic [YL99] aims at dealing with lack of precision or uncertainty. In this sense, it is perhaps closer in spirit to the human thinking than traditional logic systems. Not surprisingly, fuzzy approaches are exploited as a key technology in specific application areas, e.g., the selection of manufacturing technologies [GG12], and industrial processes where the control via conventional methods suffers from the lack of quantitative data about I/O relations. There, a fuzzy logic controller effectively synthesises an automatic control strategy from a linguistic control strategy based on an expert's knowledge.

Answer set programming (ASP) and *constraint logic programming* (CLP) are the two main logical paradigms for dealing with various classes of NP-complete combinatorial problems. ASP solvers are aimed at computing the answer sets of standard logic programs; these tools can be seen as theorem provers, or model builders, enhanced with several built-in heuristics to guide the exploration of the solution space.

Constraint logic programming (CLP) [JL87], perhaps the most natural extension of LP (or, its most relevant generalisation), has evolved over the years into a powerful programming paradigm, widely used to model and solve hard real-life problems [Ros00] in diverse application domains—from circuit verification to scheduling, resource allocation, timetabling, control systems, etc. CLP technologies can be seen as complementary to *operation research* (OR) techniques: while OR is often the only way to find the optimal solution, CLP provides generality, together with a high-level modelling environment, search control, compactness of the problem representation, constraint propagation, and fast methods to achieve a valuable solution [RVBW08].

Non-monotonic reasoning means to face the basic objection [Min75] that logic could not represent knowledge and commonsense reasoning as humans because the human reasoning is inherently *non monotonic*—that is, consequences are not always preserved, in contrast to first-order logic. Since then, a family of approaches have been developed to suit specific needs—among these, *default reasoning* [Rei80], *defeasible reasoning* [Pol87], *abstract argumentation* theory [BDKT97]. Defeasible reasoning, in particular, is widely adopted in *AI & law* applications, to represent the complex intertwining of legal norms, often overlapping among each other, possibly from different, non-coherent sources. Abstract argumentation theory, in its turn, is concerned with the formalisation and implementation of methods for rationally resolving disagreements, providing a general approach for modelling conflicts between arguments, and a semantics to establish if an argument can be acceptable or not.

Belief-desire-intention (BDI) logic is a kind of modal logic used for formalising, validating, and designing cognitive *agents*—typically, in the *multi-agent systems* (MAS) context. A cognitive agent is an entity consisting of *(i)* a belief base storing the agent's *beliefs*, i.e. what the agent knows about the world, itself, and other agents; *(ii)* a set of *desires* (or goals), i.e. the proprieties of the world the agent wants to eventually become true; *(iii)* a *plan* library, encapsulating the agent's procedural knowledge (in the form of plans) aimed at making some goals become true; and *(iv)* a set of *intentions*, storing the states of the plans the agent is currently enacting as an attempt to satisfy some desires. All such data usually consist of first-order formulas. Then, the dynamic behaviour of a BDI agent is driven by either internal (updates to the belief-base or changes in the set of desires) or external (perceptions or messages coming from the outside) events, which may cause new intentions to be created, or current intentions to be dropped. By suitably capturing the revision of beliefs, and supporting the concurrent execution of goal-oriented computations, BDI architectures overcome critical issues of "classical" logic-based technologies – *concurrency* and *mutability* – in a sound way. Overall, BDI architecture leads to a clear and intuitive design, where the underlying BDI logic provides for the formal background. Among the frameworks rooted on a BDI approach, let us mention the AgentSpeak(L) [Rao96] abstract language and its major implementation, namely Jason, Structured Circuit Semantics [LD94], Act

Plan Interlingua [Hub99], JACK [HRHL01], and dMARS—a platform for building complex, distributed, time-critical systems in C++ [dKL98].

In the reminder of this section we focus on logic programming as the most common means to endow computational agent with reasoning capabilities. Before doing so, however, we recall major definitions and notations concerning logic inference as a means for manipulating knowledge.

5.1.1 Symbolic Inference

Within the realm of symbolic AI and, in particular, computational logic, inference is the process of mechanically producing new knowledge by applying rules to some knowledge base. Such rules are commonly expressed via a particular notation, heavily leveraging on the notion of *unification*.

Substitutions and Unification. Unification [MM82] is among the most fundamental mechanism in CL: it enables the formalisation of inference, as well virtually any other symbolic manipulation of logic formulæ.

Informally speaking, unification aims at computing a *unifier* among any two FOL formulæ, i.e. a substitution (a.k.a. assignments of variables) making the two formulæ syntactically equal, by properly assigning the variables therein contained. So, in other words, unification computes substitutions out of logic formulæ, checking whether they can be made equal, or failing otherwise.

We denote substitutions as sets of mappings of the form $\sigma = \{X \mapsto \phi, X' \mapsto \phi', X'' \mapsto \phi'', \ldots\}$, where X, X', X'' are variables, and ϕ, ϕ', ϕ'' are FOL formulæ. Furthermore, we enumerate substitutions by σ or ω. Finally, we let the binary operator $(\cdot)$ denote the application of a substitution to either a formula or another substitution. So, for instance $\phi \cdot \sigma$ denotes the formula attained by applying all variable assignments carried by σ to ϕ. Similarly, $\sigma \cdot \omega$ denotes the substitution attained by applying ω to the right-hand-side of all variable assignments in σ.

Consider for instance the case of the general clause ϕ and its particular case ψ

$$\phi \equiv (g(X, Y) \leftarrow f(X, Z), f(Z, Y))$$
$$\psi \equiv (g(\mathtt{a}, \mathtt{b}) \leftarrow f(\mathtt{a}, \mathtt{c}), f(\mathtt{c}, \mathtt{b}))$$

The two clauses can be made syntactically equal via the substitution $\sigma = \{X \mapsto \mathtt{a}, Y \mapsto \mathtt{b}, Z \mapsto \mathtt{c}\}$, because $\psi = \phi \cdot \sigma$.

Accordingly, a unifier among any two non-ground FOL formulæ ϕ and ψ, is defined as a substitution σ such that $\phi = \psi \cdot \sigma$. A trivial way to recursively compute a unifier among ϕ and ψ is as follows:

$$\text{unify}(\phi,\psi) = \begin{cases} \varnothing & \text{if } \phi = \psi = \mathtt{x} \\ \{X \mapsto Y\} & \text{if } \phi = X \wedge \psi = Y \\ \bigcup_i^N \text{unify}(\alpha_i, \alpha_i') & \text{if } \phi = f(\alpha_1, \ldots, \alpha_N) \\ & \qquad \wedge\ \psi = f(\alpha_1', \ldots, \alpha_N') \\ \text{unify}(\alpha_1, \alpha_1') \cup \text{unify}(\alpha_2, \alpha_2') & \text{if } \phi = \alpha_1 \odot \alpha_2 \wedge \psi = \alpha_1' \odot \alpha_2' \\ \square & \text{otherwise} \end{cases} \tag{5.1}$$

where $\square$ denotes the lack of any unifier – capturing the situation where two formulæ cannot be unified – and $\varnothing$ denotes the empty substitution – characterising the situation where two formulæ are identical. There, f is either a predicate or functor symbol (N-ary, in both cases), α_i, α_i' are arbitrary formulæ, and $\odot$ denotes any binary logical connective.

More precisely, unification aims at computing the *most general* unifier (MGU), i.e. the unifier σ^* such that, for each substitution σ making ϕ and ψ syntactically equal (i.e., $\phi = \psi \cdot \sigma$) there exists a substitution ω making it possible to write σ as a particular case of σ^* (i.e., $\sigma = \sigma^* \cdot \omega$). Despite computing the MGU among any two formulæ is a non-trivial problem in general, an efficient algorithm is described in [MM82]. A formal description of this algorithm lays outside this chapter. However, in what follows we denote by mgu(ϕ, ψ) the function computing the MGU among any two FOL formulæ.

Inference Rules. Inference rules are functions mapping premises (logic formulæ) into conclusions (other logic formulæ). They can be denoted both as $\phi_1, \ldots, \phi_N \vdash \psi_1, \ldots, \psi_M$ or as

$$\frac{\phi_1, \ldots, \phi_N}{\psi_1, \ldots, \psi_M}\ [\text{Rule name}]$$

where $\phi_1, \ldots, \phi_N$ are premises and $\psi_1, \ldots, \psi_M$ are conclusions. Both expressions can be read as "when all $\phi_1, \ldots, \phi_N$ are known to hold, then all $\psi_1, \ldots, \psi_M$ can be inferred".

The notation if often abused by only including among the premises those formulæ which are strictly needed to draw conclusions. When this is the case, a knowledge base $\mathbf{K}$ is then assumed, behind the scenes. Thus, rules of the form $\phi \vdash \psi$ are usually written as concise notation for $\exists \sigma$ s.t. $\phi \cdot \sigma \in \mathbf{K} \vdash \psi \cdot \sigma$.

A plethora of inference rules have been defined in the history of logic. Here we focus on four major examples, corresponding to as many inference principles, namely:

$$\frac{\alpha \rightarrow \beta, \alpha}{\beta} \text{ [Modus Ponens]} \qquad \frac{\phi \cdot \sigma_1, \ldots, \phi \cdot \sigma_n}{\phi} \text{ [Subsumption]}$$

$$\frac{\alpha \rightarrow \beta, \neg\beta}{\neg\alpha} \text{ [Modus Tollens]} \qquad \frac{\alpha \rightarrow \beta, \beta}{\alpha} \text{ [Abduction]}$$

Modus ponens (resp. tollens) is a *deductive* inference rule, stating that whenever one knows that α implies β and α is true (resp. β is false), then they can infer β (resp. $\neg\alpha$). As a deductive rule, it simply elicits particular consequences which are implicit into the general premises, and therefore certain. It is for instance by modus ponens that one can infer $\mathit{grandparent}(\texttt{abraham}, \texttt{jacob})$ from a knowledge base containing the clauses $\mathit{grandparent}(X, Y) \leftarrow \mathit{parent}(X, Z), \mathit{parent}(Z, Y)$, $\mathit{parent}(\texttt{abraham}, \texttt{isaac})$, and $\mathit{parent}(\texttt{isaac}, \texttt{jacob})$.

Subsumption is an *inductive* inference rule, stating that if a number of particular expressions share the same form, than that form can be raised to general knowledge. As an inductive rule, it attempts to derive novel hypotheses out of prior experience, therefore producing uncertain conclusions that may be eventually contradicted by some later experience. It is for instance by subsumption that one can hypothesise how a general rule for grandparenthood works (i.e. $\mathit{grandparent}(X, Y) \leftarrow \mathit{parent}(X, Z), \mathit{parent}(Z, Y)$) if they only know about a number of parenthood and grandparenthood relations—namely $\mathit{parent}(\texttt{abraham}, \texttt{isaac})$, $\mathit{parent}(\texttt{isaac}, \texttt{jacob})$, and $\mathit{grandparent}(\texttt{abraham}, \texttt{jacob})$.

Finally, abduction is an *abductive* inference rule, stating that if one knows how a particular cause–effect phenomenon works, and they observe the effect, then they can infer the cause. Similarly to induction, abduction produces uncertain – yet likely – conclusions out of prior experience and some basic knowledge about the world. It is for instance by abduction that one can hypothesise that it is raining (rain) after observing that the floor is wet ($\mathit{wet\_floor}$) knowing that the flow may be wet because of either the rain or a broken glass ($\{\mathit{wet\_floor} \leftarrow \mathit{rain}, \mathit{wet\_floor} \leftarrow \mathit{broken\_glass}\}$).

Notice that only deductive rules lead to certainly correct conclusions, whereas inductive and deductive rules do not. Accordingly, Bayesian inference may be better suited to represent inductive or abductive inference – as it let us infer not only hypotheses but their likelihood as well –, yet it requires trepassing the sub-symbolic realm—as probabilities must be explicitly represented.

Inference Procedures. Inference rules alone are able to express reasoning, but they are not enough to let computational draw inferences autonomously. When dealing with inference, computational agents need an inference *procedure*, i.e. an algorithm applying one or more inference rules to a knowledge base, in such a way that conclusions are eventually reached when the algorithm terminates.

In practice, inference procedures should support the efficient and effective computation of correct conclusions. Within this scope, "efficiently" means "in useful time", whereas "effectively" refers to notable properties such *soundness* and *completeness*. There, a *sound* procedure ensures that all output conclusions are correct, whereas a *complete* procedure ensures that (at least) all correct conclusions are drawn. Notably, these properties can – and usually are – satisfied by algorithms supporting *deductive* inference procedures, whereas they are out of reach for inductive or abductive algorithms (because of uncertainty).

Another common way of categorising inference procedures is w.r.t. the *verse* they follow in applying inference rules. Two strategies may be followed by inference procedures, namely either *forward* or *backward*-chaining.

In forward-chaining, inference rules are naturally applied from premises to conclusions, in top-down fashion. The starting point is commonly a knowledge base, and, as a result, many possible conclusions are drawn—possibly even more than needed.

Conversely, in backward-chaining, inference rules are reversely applied from conclusion to premises in order to prove a particular *goal*. So, inference proceeds in a bottom-up fashion, attempting to reach the knowledge base from the goal. Under such setting, inference rules should be read in the opposite way. For instance, modus ponens should be read as "to infer β, knowing that $\alpha \rightarrow \beta$, one should prove α first".

In the reminder of this section, we briefly overview the field of logic programming, where logic and computational aspects are deeply intertwined to endow computational agents with actual automated reasoning capabilities.

5.1.2 Logic Programming

Logic programming (LP) is computational paradigm where logic is used to represent both data and programs, and inference is used to perform computations. From a human-centred perspective, LP is a means for programming computers to perform symbolic AI related tasks. However, from an agent-oriented perspective, LP is the means to endow software agents with automated reasoning capabilities.

There are a few peculiar milestones in the history of LP. These are *(i)* Robinson's proposal of the selective linear (SL) resolution principle [Rob65] for FOL, *(ii)* Kowalsky's proposal of the selective linear *definite* (SLD) resolution principle [vEK76] for Horn clauses, *(iii)* Colmerauer's proposal of Prolog [CR93], *(iv)* Clark's proposal of negation as failure (NaF) [Cla77] as an extension to the SLD principle.

Robinson's selective linear resolution principle. The SL resolution principle [Rob65] is a general procedure for proving a set of FOL formulæ in Skolemized form[3] true, by *refutation*. More precisely, the SL procedure aims at proving a set of clauses as contradictory, by attempting to derive the empty clause $\perp$ (denoting contradiction) out of it via a number of reduction steps. Thus, in practice, proving a formula ϕ true is achieved by proving $\neg\phi$ as false—i.e. by refuting it. The basic inference rule operating in this context states that, in a set of Skolemized formulæ $\mathbf{K}$ including ϕ and ψ, those formulæ may be reduced ϕ' and ψ', under the following condition:

$$\frac{x \in \phi \quad \neg y \in \psi \quad \sigma = \mathrm{mgu}(x, y)}{\phi' = (\phi - x) \cdot \sigma \quad \psi' = (\psi - \neg y) \cdot \sigma}[\mathrm{SL}]$$

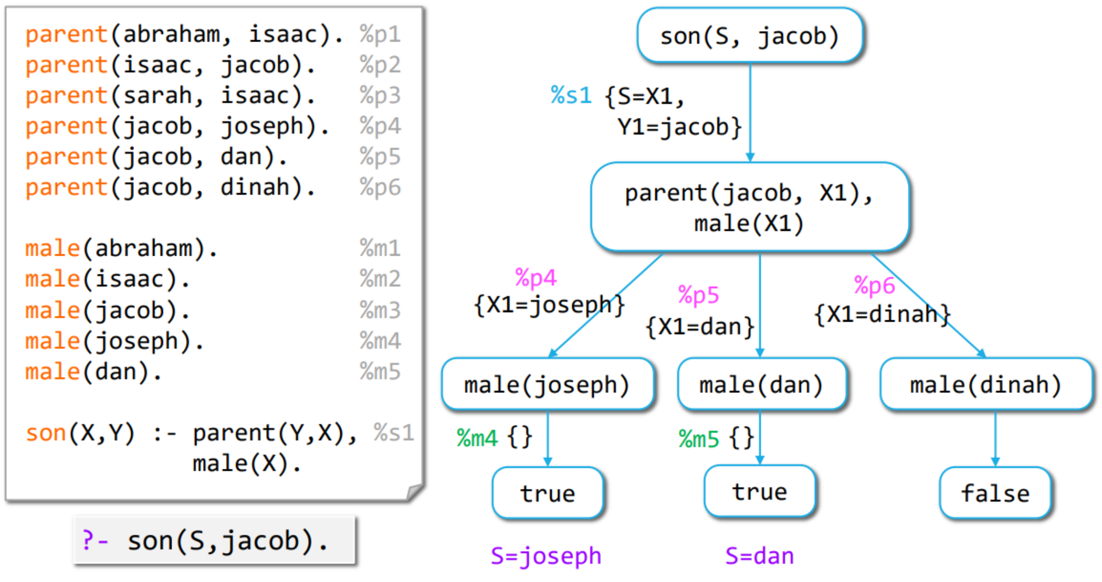
where x and y are literals possibly contained into ϕ and ψ, and operator $(-)$ denotes the eviction of a literal from a formula. In other words, if any two formulæ in $\mathbf{K}$ respectively include two literals, x and y, which can be unified by σ and such that *only* one is negated, then the two literals are evicted from the corresponding formulæ, and the substitution σ is be applied to the reminder of those formulæ. If the repeated application of this rule leads to a situation where a formula has no more literals, the original set of formulæ $\mathbf{K}$ is proved to be false, as it contains an unsatisfiable (i.e. contradictory) formula—namely, the one that has been emptied by the SL procedure. Otherwise, if no further reduction is possible, $\mathbf{K}$ is proved to *satisfiable*.

Backward-chaining can be performed via SL. This implies a goal γ can be proved against a knowledge base $\mathbf{K}$. When this is the case, proof by refutation if applied to the set $\neg\gamma \cup \mathbf{K}$: at each step, some literal of $\neg\gamma$ is reduced, along with some other formula in $\mathbf{K}$. Upon termination, if all literals of $\neg\gamma$ have been evicted, then $\neg\gamma$ is considered unsatisfiable, therefore γ is provable, and the formula $\gamma \cdot \sigma_1 \cdot \sigma_2 \cdots$ – attained by applying all the MGU computed at each reduction step – represents a *solution* for γ, according to $\mathbf{K}$.

The whole process subtends the so-called *proof tree*: at each step of the procedure, several formualæ may be selected from $\mathbf{K}$, therefore the algorithm may follow as many different paths. In other words, the SL resolution is a *non-deterministic* algorithm. Details and examples are provided for instance in [CB15].

Kowalsky's selective linear definite resolution principle. The SLD resolution principle [vEK76] is a refinement of the SL principle, tailored on Horn clauses. By restricting the scope of resolution to Horn logic, Kowalsky shows how logic resolution can be described via a *procedural* interpretation, mimicking programs execution. The result is a non-deterministic algorithm describing how the proof tree of any given query – expressed as a goal – against any given knowledge base – expressed as a set of definite clauses – can be *lazily* constructed, while attempting to refute the query.

[3]`cf.https://mathworld.wolfram.com/SkolemizedForm.html`



```
parent(abraham, isaac).  %p1
parent(isaac, jacob).    %p2
parent(sarah, isaac).    %p3
parent(jacob, joseph).   %p4
parent(jacob, dan).      %p5
parent(jacob, dinah).    %p6

male(abraham).           %m1
male(isaac).             %m2
male(jacob).             %m3
male(joseph).            %m4
male(dan).               %m5

son(X,Y) :- parent(Y,X), %s1
            male(X).
```

```
?- son(S,jacob).
```

Figure 5.1: An example of proof tree generated by the SLD resolution principle while attempting to prove the goal $son(S, \texttt{jacob})$ against the depicted knowledge base.

The basic inference rule operating in this contexts states that any goal γ can be rewritten as γ', provided that some definite clause ϕ exists in the knowledge base whose head unifies with some literal in γ. When this is the case, the matching literal of γ is replaced by the body of ϕ—or simply removed in case ϕ is a fact. More formally:

$$\frac{\gamma \equiv (\leftarrow g_1, \ldots, g_i, \ldots, g_n) \qquad \phi \equiv (h \leftarrow b_1, \ldots, b_m) \qquad \sigma = \text{mgu}(g_i, h)}{\gamma' \equiv (\leftarrow g_1, \ldots, g_{i-1}, b_1, \ldots, b_m, g_{i+1}, \ldots, g_n) \cdot \sigma}[\text{SLD}]$$

The recursive application of this rule is what enables automated reasoning. Similarly to the SL case, if γ can be emptied via a number of successive applications of the inference rule, then it is considered proven by refutation.

The relation among SL and SLD becomes quite evident if one writes clauses in disjunctive form. Under such perspective, γ is written as a disjunctive of negated literals $\neg g_1 \vee \ldots \vee \neg g_n$, as well as each rule $\phi \equiv (h \vee \neg b_1 \vee \ldots \vee \neg b_m)$. Therefore, whenever applying the SL inference rule to any literal in γ, one can only choose among the heads of each rule, as all literals in γ are negated and all heads of all rules are positive.

Figure 5.1 exemplifies the proof tree generated by the recursive application of SLD inference rule to the goal $son(S, \texttt{jacob})$. The resolution attempts to prove the goal against a knowledge base describing Abraham's family tree, represented in the same image. Each node in the tree represents a rewritten form of the original goal, whereas each arc subtends the selection of a rule from the knowledge base and a literal from the source node: the label of each edge describes their MGU.

Of course, at each step of the SLD resolution, several choices may be taken. For instance, several literals may be selected, and each literal may unify with potentially many rules in the knowledge base. The theoretical formulation of the SLD principle addresses such choices via *non-determinism.* However, whenever an actual computational agent needs to perform automated reasoning, it must leverage upon some smart strategy to explore the proof tree sequentially, and in useful time.

Colmerauer's Prolog. Prolog [CR93] is the most successful expression of the LP paradigm. It is at the same time a particular way of expressing logic resolution, and a very powerful technology to perform automated reasoning in practice. Here we focus on the theoretical aspects of Prolog, whereas technological aspects are treated later in this thesis.

Prolog is essentially a particular way to make the exploration of the proof tree subtended by the SLD resolution principle *sequential.* In other words, Prolog makes the SLD resolution a sequential algorithm. More precisely, Prolog assumes goals to be *ordered* disjunctions of literals (left to right), and knowledge bases to be *ordered* sets of definite clauses (top to bottom). Under such assumption, it adopts the following strategy while dealing with the non-determinism of SLD resolution:

- literals in the current goal are reduced from left to right,
- rules are selected from top to bottom.

This strategy subtends a *depth-first* exploration of the proof tree.

Another relevant modelling choice in Prolog concerns the syntax for predicates and terms. In fact, Prolog represents predicates and structured terms in the same way, collapsing their syntaxes and making the two notions interchangeable. This simplifies the definition of meta-predicates – i.e. predicates accepting other predicates as arguments –, as terms can be used to represent predicates. Meta-predicates, in turn, make Prolog's semantics very flexible, as disjunctions, implications, negations, and many other features of FOL which are not supported by Horn clauses can be re-introduced in Prolog in this way.

Clark's Negation as Failure. NaF [Cla77] is the last relevant extension of SLD we discuss in this section. It essentially aims at supporting negation in Horn clauses. SLD resolution extended with NaF is often concisely referred to as SLDNF.

SLDNF supports negation by defining the meta-predicate $not(g)$ which is proven true if and only if the argument goal g is proven unsatisfiable. So, in procedural terms, the goal $not(g)$ can be read as "attempt to prove g, and, if no solution exists, then $not(g)$ is true, false otherwise".

Notably, NaF subtends a *closed* world assumption, where everything that is not know – nor deducible from what is known – is false.

5.2 Sub-symbolic Reasoning

This section contains contributions from the following works of ours: [CCO20]

Within the sub-symbolic realm, the term 'inference' is often abused. There, 'inference' refers to the application phase of any data-driven solution which has been previously trained on data. Therefore, inference in simply a phase in the life-cycle of a data-driven model, as opposed to training.

In the inference phase of any sub-symbolic system, novel information is actually drawn, and that information strongly depends on what the systems has learned from data (prior knowledge) during training. For instance, classifiers enable the labelling of arbitrarily complex unknown data according to some predefined set of labels. Similarly, regressors enable predictions on unknown data, out of prior experience. So, in a broad sense, sub-symbolic systems support the inference of novel, useful information. By definition, however, no symbolic manipulation of data occurs in sub-symbolic systems, regardless of whether training or inference are considered. Hence, it is cumbersome speaking of reasoning.

Nevertheless, a number of recent proposals are pushing relevant aspects of logic inference into the sub-symbolic realm. As a result, methods for building *hybrid* systems – i.e. systems mixing symbolic and sub-symbolic means to represent, learn, or infer – are flourishing.

Generally speaking, hybridisation may occur in two ways, namely by *model integration* or by *symbolic knowledge embedding*. In the former case, symbolic information is used to *structure* or *constrain* the behaviour of a sub-symbolic system—in most cases, a neural network, because of its malleability. In the latter case, symbolic information is *embedded* into a sub-symbolic representation to enable its sub-symbolic processing.

5.2.1 Model Integration

In this category we review the main attempts to *integrate* symbolic models (such as the logic ones) with sub-symbolic ones (such as statistical and numerical). The main research lines here are those related to the neural-symbolic computing [HHH07] – the study of logics and connectionism as well as statistical approaches working on the integration of computational learning and symbolic reasoning – and relational learning [DR08a]—focused on learning expressive logic / relational representations.

Approaches in this category integrate logic and symbolic knowledge with sub-symbolic predictors such as (deep) neural networks. Integration exploits logic rules expressed via FOL – or some subset of it – which are used to either constrain or structure the behaviour of one or more predictors.

On the one hand, constraining is commonly performed by extending the loss function used by most numeric learning algorithms – there including the back-propagation algorithm used for neural networks – with an additive, regularisation term constructed from the logic constraints. The numeric predictor is then trained "as usual", via optimisation—i.e. minimising some loss function. However, thanks to a *regularisation* term attained by encoding the logic rules accordingly, the training process is more likely to select a set of parameters for the numeric predictor, which are consistent with those logic rules. Generally speaking, the key advantage of these approaches lies in the blended integration of different models, where the logic one – where expressing crisp information is trivial – can be used to inject prior knowledge or common-sense into the sub-symbolic one, even in lack of data.

On the other hand, structuring is commonly performed by building the sub-symbolic predictors in such a way that their internal structure mirrors the provided symbolic knowledge. For this reason, malleable models such as neural networks are often preferred to serve this purpose. There, the internal topology of neurons, as well as their activation functions, are structured in such a way to mimic some relevant property of given logic rules—such as their interpretation. Computation is then shifted into the sub-symbolic realm, since the so-constructed predictors act ordinarily. Generally speaking, the key advantage of these approaches sub-symbolic emulation of symbolic facilities, thus allowing efficiency of reasoning in particular cases – at the price of constraining it scope of application –, and robustness w.r.t. missing or contradictory data.

Logic as constraints. Paradigmatic works in this category are for instance: DNN with Logic Rules [HML$^+$16], Logic Tensor Networks (LTN) [SG16, SDG17], Semantic Loss Function (SLF) [XZF$^+$18], and Lyrics [MGDG19].

DNN with Logic Rules (no concise name is given) [HML$^+$16], proposes a method for constraining a (deep) neural network behaviour via FOL rules. The proposed framework enables neural networks to be simultaneously trained on labelled data or logic rules, via an iterative distillation procedure aimed at transferring the symbolic knowledge encoded in the logic rules into the network parameters. To do so, the authors propose the exploitation of two networks: a *teacher* and a *student* one. The teacher network is rule-regularised via an *ad-hoc* term added to the loss function, meaning that it is trained by keeping into account the user-provided logic rules. In particular, logic constraints are encoded into the loss function via soft logic [BBHG17]. Conversely, the student network is trained to balance between emulating the teacher network output and predicting the expected outcomes of the dataset.

LTN [SG16, SDG17] integrate learning based on tensor networks [SCMN13] with reasoning based first-order many-valued logic [Ber08]. They enable a range of knowledge-based tasks using rich knowledge representation in FOL to be combined with efficient data-driven machine learning based on the manipulation of real-valued vectors. Notably, integration is defined upon the Real Logic [SDG17]. FOL formulæ are used to build a loss function that aims at training a network capable of approximating the truth value (in the $[0, 1]$ interval) of the formulæ given as input. This is done by searching for the best possible representation for symbolic constructs in a vector space (grounding of atoms, functions, predicates), so that the satisfiability of the network is as close as possible to 1 on the test dataset. The resulting network is able to learning from the rightly-labelled real examples, but keeps the logic imprint given in the training phase.

SLF [XZF+18] is another attempt of bridging neural networks and symbolic constraints via loss-function manipulation, similarly to LTN. In the intentions of its authors, it aims to

- improve the predictive performance of neural networks – by allowing the training process to take background knowledge into account –, and
- support semi-supervised learning.

To do so, SLF constrains the training process of a neural network via some *propositional* logic formulæ which are then encoded as part the loss function exploited by the training algorithm. Such formulæ consist of boolean variables representing input and output neurons of the networks to be constrained, possibly combined via classical logic connectors.

Finally, Lyrics [MGDG19] is an extension of LTN, improving the way symbolic knowledge is declaratively enforced while training the sub-symbolic part of an intelligent system. According to the authors, the major applications of Lyrics are related to predictive model verification, semi-supervised learning with background knowledge, collective classification [SNB+08], and text chunking. Similarly to LTN, Lyrics can combine one or more neural networks into a single computational graph. Each neural network is mapped onto a logic predicate, when necessary, while (possibly global) constrains over the outcomes of the networks are mapped into logic formulæ. The resulting computational graph is then optimised against the available data via state-of-the-art gradient-descent technologies—e.g. TensorFlow.

Logic as structure. Paradigmatic works in this category are for instance: Knowledge-Based Artificial Neural Networks (KBANN) [TSN90], CILP++ [FZdG14], Neural Theorem Prover (NTP) [RR17], Differentiable Inductive Logic Programming (∂ILP) [EG18], DeepProbLog [MDK$^+$18], and Lifted Relational Neural Networks (LRNN) [SAZ$^+$18].

KBANN [TSN90] is one of the earliest attempts of exploiting symbolic AI to govern the structure and the behaviour of neural networks. It is capable of devising the structure of a neural network from a symbolic knowledge base containing the user-defined, symbolic background knowledge. More precisely, KBANN assumes a stratified, Prolog-like, logic theory is available, encoding the background knowledge. Under this assumption, the KBANN algorithm aims at creating a neural network semantically reflecting the symbolic knowledge from which it was created. This step essentially sets the network structure and weights in order to reflect the rules contained into the logic theory. The resulting neural network can then be trained over data via back-propagation, in order to refine or generalise its functioning over (possibly novel) data.

CILP++ [FZdG14] is a model aimed at performing inductive logic programming (ILP) via bottom clause propositionalisation and neural networks. CILP++ leverages on *(i)* neural networks to make ILP faster, and on *(ii)* propositionalisation to make the construction of neural networks out of arbitrary logic theories possible. More precisely, propositionalisation [Lac10] is a preliminary step, which is necessary to convert the example clauses into real vectors and the background knowledge into a multi-layered neural network to be fed with those vectors. Of course, the structure of this network reflects the rules contained in the background knowledge, and the input layer contains a neuron for each possible atom used in the background knowledge.

NTP [RR17] are neural networks acting as logic reasoners (a.k.a. theorem provers). They are built by taking inspiration from backward-chaining-based reasoning algorithms, as in Prolog. In particular, the neural network is recursively constructed to encapsulate the knowledge encoded in some logic theory, and trained to correctly answer to all possible queries on such theory. Of course, the structure of the resulting network reflects the structure of the clauses contained into the source logic theory. However, differently from the other techniques presented in this sub-category, both theories and queries supported by NTP can contain logic variables, as NTP is able to calculate, at the neural network level – i.e., in the sub-symbolic model –, the logic unification. In other words, NTP perform symbolic reasoning on top of sub-symbolic and distributed representations of knowledge.

∂ILP [EG18] is another means for ILP leveraging on neural networks. It works by mimicking logic deduction on definite clauses via a neural network, similarly to NTP. However, differently from NTP, ∂ILP perform deduction using forward chaining, instead of backward chaining. Briefly speaking, the authors re-interpret ILP as a binary-classification problem. As for other similar approaches discussed in this category, a neural network is constructed in such a way that its structure reflects a grounded version of the background knowledge. The resulting network is then trained to minimise the cross-entropy with respect to positive and negative examples.

DeepProbLog [MDK+18] is another attempt of blending neural networks with logic programming, and in particular *probabilistic* logic programming (PLP). It is an extension of ProbLog exploiting neural networks for *(i)* computing the probabilities of facts, and *(ii)* letting neural classifiers be used as logic predicates—defined as "neural predicates" by the authors. In particular, each DeepProbLog program is translated into a tensorial computational graph – possibly including one or more neural classifiers as sub-graphs – to be optimised via gradient descend. The structure of the computational graph reflects the structure of the rules contained into the DeepProbLog program. The optimisation step is aimed at simultaneously setting all the possible parameters regulating the behaviour of the computational graph, including the probabilities of facts and the internal weights of neural predicates. The resulting sub-symbolic system is then exploited to draw probabilistic inferences. In other words, hybrid systems based on DeepProbLog fruitfully combine probabilistic reasoning and sub-symbolic classification in a single, unified, coherent framework.

Finally, LRNN [SAZ+18] aim at performing relational learning from data via neural networks. Similarly to DeepProbLog, LRNN exploit sets of weighted first-order formulæ as structural templates for building a neural network to be trained over the available data. The resulting network is exploited to infer latent rules buried in data and to estimate the weights of the existing clauses.

5.2.2 Symbolic Knowledge Embedding

In this category we review the main attempts to *embed* symbolic (and, in particular, logic) knowledge into arrays of numbers, to make them amenable of sub-symbolic processing.

Most techniques developed so far are tailored on *description logics*, that are particular sub-sets of FOL generally aimed at describing categories of entities and their possible relations, along with instances of both. The key idea is to translate components of an ontology (a.k.a. knowledge graph, or simply KG) into continuous vector spaces, to allow neural networks to accept such a type of structured information as input and take advantage of its background knowledge to perform ordinary machine learning tasks.

Most of the currently-available techniques perform the embedding task only on the basis of observed facts. Given a KG, knowledge graph injection techniques first represent entities and relations in a continuous vector space, and then measure facts plausibility exploiting some scoring function. Entity and relation embeddings can be obtained by maximising the total plausibility of observed facts.

During this whole procedure, the learned embeddings are only required to be compatible within each individual fact, and hence might not be predictive enough for downstream tasks [WWG15, WZL$^+$15]. As a result, more and more researchers have started to add other types of information, including logic rules [WWG15, RSR15, GWW$^+$16], in order to learn more predictive embeddings.

The noteworthy approaches that we deem significant for the purpose of this survey – as they combine symbolic and sub-symbolic models – are:

- Rescal + Trescal (2015) [WWG15]
- Ins (2015) [WZL$^+$15]
- Low-rank Logic Embeddings, LLE (2015) [RSR15]
- Kale (2016) [GWW$^+$16]
- Oscar (2019) [GDF19]

In particular, [WWG15, WZL$^+$15] exploit rules to refine embedding models aimed at KG completion. KG completion is formulated as an integer linear programming problem, where the objective function is generated from embedding models and constraints are generated from rules. Facts inferred in this way are the most preferred by the embedding models and comply with all the rules. By incorporating rules, these approaches can greatly reduce the solution space and significantly improve the inference accuracy of embedding models. Trescal [WWG15] is an extension of Rescal, requiring the arguments of a relation to be entities of certain specified types.

Along this line, other works – e.g., [RSR15, GWW$^+$16] – propose approaches that embed KG facts and logic rules simultaneously in a unified framework. In particular, in Ins, formulæ are injected into the embeddings of relations and entity-pairs, i.e., the embeddings are estimated such that predictions based on them conform to given logic formulæ. Kale, on the other side, represents rules as complex formulæ modelled by t-norm fuzzy logics. Embedding then amounts to minimising a global loss over both atomic and complex formulae. Thus embeddings are learnt as compatible with rules.

In [GDF19] the authors propose a method, Oscar, for injecting task-agnostic knowledge from a KG into a neural network during the training. Oscar is a pre-training regularisation technique capable of injecting world knowledge and ontological relationships into a deep neural network: the expert knowledge is exploited as a regulariser for the network.

It is worth noting that in all these approaches rules are modelled separately from embedding models, serving as post-processing steps: this is why we classify these work as combination and not integration. Furthermore, all these works share a common drawback, in that they have to instantiate universally-quantified rules into ground rules before learning their models. This is called grounding procedure, and can be time- and space-inefficient—especially when dealing with big data scenarios or in case of rules complexity.

5.2.3 Hybrid Systems: Final Remarks

Hybrid systems are still in their infancy. Hybridisation usually comes at the cost of a reduced expressiveness of the logic formalism adopted. Empirically, we observe that full fledged FOL is too complex to handle for sub-symbolic systems, which are therefore forced to take countermeasures into account. Once again, issues concern KR. In particular, *intensional* logic representations as well as the flexibility of logic – which is open to the addition of novel symbols –, are what makes hybridisation difficult.

Countermeasures generally involve using a sub-sets of FOL – thus renouncing to the full expressiveness of FOL –, and enforcing knowledge to be extensively represented.

In the former case, constraints are imposed at the KR level. These may involve *(i)* forbidding structured terms, *(ii)* limiting the shape of predicates or clauses, *(iii)* focussing on simpler logics such as Horn logic, description logics, or predicate logic, *(iv)* or a combination of these constraints. In all such cases, the scope of the hybrid system is reduced, w.r.t. a full-fledged symbolic system. Consider for instance the case of Horn logic with no structured terms. This would imply, for instance, that lists could not be used in the symbolic part of the system—hence greatly reducing the practical reach of hybrid systems.

In the latter case, *grounding* of the knowledge base is usually assumed as a preliminary step before any sub-symbolic processing. In the general case, grounding a non-ground knowledge base involves enumerating all the possible variable assignments of for all variables of all formulæ therein contained. Variable assignments may in turn be infinite, depending on how may items the underlying Herbrand universe contains. Of course, grounding is only possible in practice when the amount of items in the Herbrand universe – and therefore the amount of admissible variable assignments –, is *finite*. Notably, by admitting structured terms for KR, the Herbrand universe will certainly be infinite (cf. section 3.1.1)—hence why leveraging on grounding usually subtends forbidding (at least) structured terms from the KR formalism.

Chapter 6

Explaining AI via Symbolic Knowledge

It is undeniable that AI and ML are nowadays becoming more and more intertwined with a growing number of aspects of people's every day life [Hel19, Ell19]. In fact, more and more decisions are delegated by humans to software agents whose intelligent behaviour is not the result of some skilled developer endowing them with some clever code, but rather the consequence the agents' capability of learning, planning, or inferring what to do from data.

In spite of the large adoption, intelligent machines whose behaviour is the result of automatic synthesis / learning procedures are difficult to trust for most people—in particular when they are not expert in the field. This is especially true for agents leveraging on machine or deep learning based techniques, often producing models whose internal behaviour is opaque and hard to explain for their developers too.

There, agents often tend to accumulate their knowledge into *black-box* predictive models which are trained through ML or DL. As we further discuss in this chapter, "black boxes" are models where knowledge is *sub-symbolically* represented – such as NN, support vector machines (SVM), or random forests –, and it is therefore difficult, for humans, to understand what they actually know, or what led them to a particular decision.

Such difficulty in understanding black-boxes content and functioning is what prevents people from fully trusting – and thus accepting – them. In several contexts, such as the medical or financial ones, it is not sufficient for intelligent agents to output bare decisions, since, for instance, ethical and legal issues may arise. An *explanation* for each decision is therefore often desirable, preferable, or even required. Furthermore, it may happen for instance that black-boxes *silently* learn something wrong (e.g., Google image recognition software that classified black people as gorillas [FH17, Cra16]), or something right, but in a biased way (like the "background bias" problem, causing for instance husky images to be recognised only because of their snowy background [RSG16]).

Accordingly, in the reminder of this chapter we discuss the meaning and the role of explanations in modern AI, we review the recent literature on this topic, and we present the possible means to construct *symbolic* explanations for sub-symbolic predictors.

6.1 eXplainable Artificial Intelligence

This section contains contributions from the following works of ours: [CCSO20, CSOC20]

Most intelligent systems (IS) today leverage on *numerical* predictive models which are trained from data through ML. The reason for such a wide adoption is easy to understand. We live in an era where the availability of data is unprecedented, and ML algorithms make it possible to semi-automatically detect useful statistical information hidden into such data. Information, in turn, supports decision-making, monitoring, planning, and forecasting in virtually any human activity where data is available.

However, ML is not the silver bullet. Despite the increased predictive power, ML comes with some well-known drawbacks which make it perform poorly in some use cases. One blatant example is algorithmic *opacity*—that is, essentially, the difficulty of human mind in *understanding* how ML-based IS function or compute their outputs. Such difficulty is a serious issue in all those contexts where human beings are liable for their decision or must provide some sort of *explanation* for it—even if the decision has been supported by some IS. For instance, think about a doctor willing to motivate a serious, computer-aided diagnosis, or, a bank employee in need of explaining to a customer why his/her profile is inadequate for a loan. In all contexts, ML is at the same time an enabling – as it aids the decision process by automating it – and a limiting factor—as opacity prevents human awareness of *how* the decision process works.

Opacity is why ML predictors are also referred to as *black boxes* into the literature. The "black box" expression refers to models where knowledge is not explicitly represented [Lip18]. The lack of some explicit, symbolic representation of knowledge is what makes it hard for humans to *understand* the functioning of black boxes, and why they led to suggest or undertake a given decision. Obviously, troubles in understanding black-box content and functioning prevents people from fully trusting – therefore accepting – them. To make the picture even more complex, current regulations such as the GDPR [VvdB17] are starting to recognise the citizens' *right to explanation* [GF17]—which implicitly requires IS to eventually become *understandable.* In fact, understanding IS is essential to guarantee algorithmic fairness, to identify potential bias/problems in the training data, and to ensure that IS perform as designed and expected.

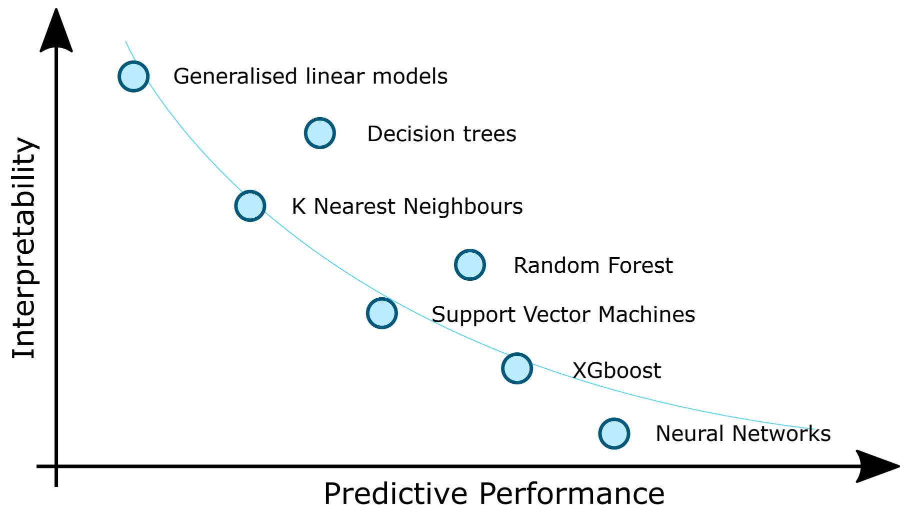


Figure 6.1: Interpretability/performance trade-off for some common sorts of black-box predictors

Unfortunately, the notion of understandability is neither standardised nor systematically assessed, yet. At the same time, there is no consensus on what exactly providing an *explanation* should mean when decisions are supported by a black box. However, several authors agree that not all black boxes are equally *interpretable*—meaning that some black boxes are easier to understand than others for our mind. For example, fig. 6.1 is a common way to illustrate the differences in black-box interpretability.

Even though informal – as pointed on in [Rud19], given the lack of way to measure "interpretability" – fig. 6.1 effectively express why more research is need on understandability. In fact, the image essentially states how the better performing black boxes are also the less interpretable ones. This is a problem in practice since only rarely predictive performances can be sacrificed in favour of a higher degree of interpretability.

To tackle such issues, the *eXplainable AI* (XAI henceforth) research field has recently emerged. Among the many authors and organisations involved in the topic, DARPA has proposed a comprehensive research road map [Gun16] which reviews the main approaches to make black boxes more understandable. There, DARPA categorises the many currently available techniques aimed at building meaningful interpretations or explanations for black-box models, it summarises the open problems and challenges, and it provides a successful reference framework for the researchers interested in the field. Unfortunately, in spite of the great effort in defining terms, objects, and methods for the research line, a clear definition of fundamental notions such as *interpretation* and *explanation* is still missing.

6.1.1 Related works

Notions such as explanation, interpretation, transparency, etc., are mentioned, introduced, or informally defined in several works. However, a coherent framework has not emerged yet.

In this subsection we recall some main contributions from the literature where the concepts of explanation and interpretation – or any variant of theirs – are discussed. Our goal here is to highlight the current lack of consensus on the meaning of such terms, for which we propose a possible, unambiguous alternative in the next sections.

Similarly to what we do here, Lipton [Lip18] starts his discussion by recognising how most definition of ML interpretability are often inconsistent and underspecified. In his clarification effort, Lipton essentially maps interpretability on the notion of *transparency*, and explanation on the notion of *post-hoc* interpretation. Then, he enumerates and describes the several possible variants of transparency, that are *(i)* simulatability – i.e., the *practical* possibility, for a human being, to "contemplate the entire model at once" and simulate its functioning on some data – which characterises, for instance, generalised linear models; *(ii)* decomposability – i.e., the possibility, for the model to be decomposed in elementary parts whose functioning is intuitively understandable for humans and helpful in understanding the whole model – which characterises, for instance, decision trees; and *(iii)* algorithmic transparency – i.e., the possibility, for a human being, to intuitively understand how a given learning algorithm, or the predictors it produces, operate – which characterises, for instance, k-nearest-neighbours techniques. Similarly, *post-hoc* interpretability is defined as an approach where some information is extracted from a black box in order to ease its understanding. Such information have not necessarily to expose the internal functioning of the black box. As stated in the paper: "examples of post-hoc interpretations include the verbal explanations produced by people or the saliency maps used to analyse deep neural networks".

Conversely, Besold et al. [BU18] discuss the notion of explanation at a fundamental level. There, the authors provide a nice philosophical overview on such topic, concluding that "explanation is an epistemological activity and explanations are an epistemological accomplishment—they satisfy a sort of epistemic longing, a desire to know something more than we currently know. Not only do they satisfy this desire to know, they also provide the explanation-seeker a direction of action that they did not previously have". Then they discuss the topic of explanation in AI from an historical perspective. In particular, when focussing on ML, they introduce the following classification of IS systems: *(i)* opaque systems – i.e., black boxes acting as oracles where the logic behind predictions is not observable or understandable –, *(ii)* interpretable systems – i.e., white boxes whose functioning is understandable to humans, also thanks to expertise, resources, or tools –, and *(iii)* comprehensible systems—i.e., "systems which emit *symbols* along with their outputs, allowing the user to relate properties of the input to the output". According to this classification, while interpretable systems can be inspected to be understood – thus letting observer draw their explanations by themselves –, comprehensible systems must explicitly provide a symbolic explanation of their functioning. The focus there is thus on *who* produces explanations, rather than *how*.

In [DVK17], interpretability of ML systems is defined as "the ability to explain or to present in understandable terms to a human". Interpretations and explanations are therefore collapsed in this work, as confirmed by the authors using the two terms interchangeably. The reminder of that paper focuses *(i)* on identifying under which circumstances interpretability is needed in ML, and *(ii)* how to assess the quality of some explanation.

The survey by Guidotti et al. [GMR$^+$19] is a nice entry point to explainable ML. It consists of an exhaustive and recent survey overviewing the main notions, goals, problems, and (sub-)categories in this field and it encompasses a taxonomy of existing approaches for "opening the black box"—which may vary a lot depending on the sort of data and the family of predictors at hand. There, the authors define the verb to interpret as the act of "providing some meaning of explaining and presenting in understandable terms some concepts", borrowing such a definition from the Merriam-Webster[1] dictionary. Consequently, they define interpretability as "the ability to explain or to provide the meaning in understandable terms to a human"—a definition they again borrow from [DVK17]. So, in this case as well the notions of *interpretation* and *explanations* are collapsed.

In [Rud19], Rudin does not explicitly provide a definition for explainability or interpretability, and she refers about interpretable or explainable ML almost interchangeably. However, she states some interesting properties about *interpretability*, which influenced our work. In particular, she acknowledges that "interpretability is a domain-specific notion". Furthermore, she links interpretability of information with its complexity – and, in particular, its *sparsity* –, as the amount of cognitive entities the human mind can at one is very limited ($\sim 7 \pm 2$ according to [Mil56]). As far as explainability is concerned, apparently, Rudin adopts a *post-hoc* perspective similar to the one in [Lip18], as she writes "an explanation is a separate model that is supposed to replicate most of the behaviour of a black box". In the reminder of the paper, the author argues how the path towards interpretable ML steps through a wider adoption of inherently interpretable predictors – such as generalised linear models or decision trees – instead of the relying on *post-hoc* explanations which do not reveal what is inside black boxes—thus preventing their full understanding.

[1]`https://www.merriam-webster.com/dictionary/interpret`

Finally, the recent article by Rosenfeld et al. [RR19] is similar in its intents to our current work. There, the authors attempt to formally define what explanation and interpretation respectively are in the case of ML-based classification. According to the authors, "interpretation" is a function mapping data, data schemes, and predictors to some representation of the predictors internal logic, whereas "explanation" is defined as "the human-centric objective for the user to understand" a predictor using the aforementioned interpretation function. Other notions are formally defined into the paper, such as for instance, *(i)* explicitness, *(ii)* fairness, *(iii)* faithfulness, *(iv)* justification, and *(v)* transparency. Such concepts are formally defined in terms of the aforementioned interpretation and explanation functions. The reminder of the paper then re-interprets the field of XAI in terms of all the notions mentioned so far.

6.2 Explanation vs. Interpretation

This section introduces the preliminary notions, intuitions, and notations we leverage upon in section 6.2.1 and subsequent sections, in order to formalise our abstract framework for agent-based explanations. We start by providing an intuition for the notion of *interpretation*, and, consequently, for the *act* of interpreting something. Accordingly, we provide an intuition for the property of "being interpretable" as well, stressing its comparative nature. Analogously to what we did with *interpretation*, we then provide intuitions for terms such as *explanation* and its derivatives.

About interpretation. Taking inspiration from the field of Logics, we define the *act* of "interpreting" some object X as the activity performed by an agent A – either human or software – assigning a *subjective* meaning to X. Such meaning is what we call *interpretation*. Roughly speaking, an object X is said to be *interpretable* for an agent A if it is *easy* for A to draw an interpretation for X—where "easy" means A requires a low *computational* (or *cognitive*) effort to understand X. For instance, consider the case of road signs, which contain symbols instead of scripts to be easily, quickly, and intuitively interpretable.

We model such intuition through a function $I_A(X) \mapsto [0, 1]$ providing a *degree of interpretability* – or simply interpretability, for short – for X, in the eyes of A. The value $I_A(X)$ is not required to be directly observable or measurable in practice, since agents' mind may be inaccessible in most cases. This is far from being an issue, since we are not actually interested in the absolute value of $I_A(X)$, for some object X, but rather we are interested in being able to order different objects w.r.t. their subjective interpretability. For instance, we write $I_A(X) > I_A(Y)$, for two objects X and Y, meaning that the former is more interpretable than the latter, according to A.

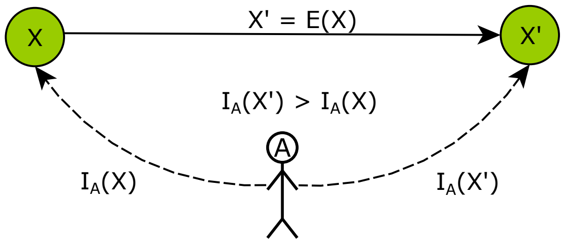


Figure 6.2: Explanation vs. Interpretation: a simple framework

For example, consider the case of a neural network and a decision tree, both trained on the same examples to solve the same problem with similar predictive performances. Both objects may be represented as graphs. However, it is likely for a human observer to see the decision tree as more interpretable—as their nodes bring semantically meaningful, high-level information.

Summarising, we stress the subjective nature of interpretations, as agents assign them to objects according to their State of Mind (SoM) [PW78] and background knowledge, and they need not be formally defined any further.

About explanation. We define "explaining" as the activity of producing a more interpretable object X' out of a less interpretable one, namely X, performed by agent A. More formally, we define *explanation* as a function $E(X) \mapsto X'$ mapping objects into other objects, possibly, in such a way that $I_A(X') > I_A(X)$, for some agent A. The simple framework described so far is summarised in fig. 6.2.

Notice that human beings tend to collapse into the concept of "explanation" the whole sequence of steps actually involving both explaining and interpreting, according to our framework. This happens because, if the explained object X' is as interpretable for the listening agent B as it is for the explaining agent A, then both A and B are likely to be satisfied with X'. Conversely, it may also happen the explanation E adopted by A produces an object X', which is more interpretable than X for A but not for B. Similarly to how two persons would handle such an unpleasant situation, we envision that interaction and communication may be adopted to break such *impasses* in multi-agent systems.

In the following sections, we develop such an idea, describing how our simple framework could be extended to support ML-based intelligent systems.

6.2.1 A conceptual framework for XAI

In AI several tasks can be reduced to a functional model $M : X \to Y$ mapping some input data $X \subseteq \mathcal{X}$ from an input domain $\mathcal{X}$ into some output data $Y \subseteq \mathcal{Y}$ from an output domain $\mathcal{Y}$.

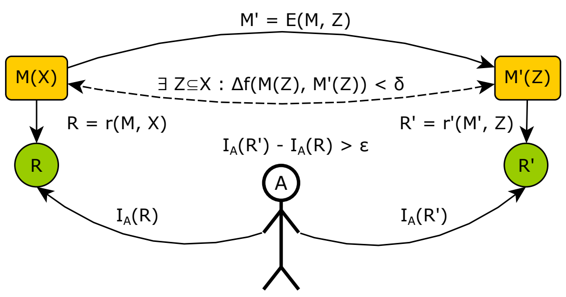


Figure 6.3: Local explanation and interpretation of a model

In the following, we denote as $\mathcal{M}$ the set of all *analogous* models $M' : X \to \mathcal{Y}$, which attempts to solve the same problem on the same input data—usually, in (possibly slightly) different ways. For instance, according to this definition, a decision tree and a neural network, both trained on the same data-set to solve the same classification problem with similar accuracies, are analogous—even if they belong to different families of predictors.

At a very high abstraction level, many tasks in AI may be devoted to compute, for instance:

- the best $M^* \in \mathcal{M}$, given $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$ (e.g. supervised ML),
- the best M^* and Y, given X (e.g. unsupervised ML),
- the best Y^*, given X and M (e.g. informed/uninformed search),
- the best X^*, given Y and M (e.g. abduction, most likely explanation), etc

according to some goodness criterion which is specific for the task at hand.

In the reminder of this section, we discuss how explanation may be defined as a function searching or building a – possibly more interpretable – model w.r.t. the one to be explained. For this process to even make sense, of course, we require the resulting model to be not only analogous to the original but also similar in the way it behaves on the same data. We formalise such a concept through the notion of *fidelity*.

Let $M, M' \in \mathcal{M}$ be two analogous models. We then say M has a *locally* good *fidelity* w.r.t. M' and Z if and only if $\Delta f(M(Z), M'(Z)) < \delta$ for some arbitrarily small threshold $\delta \geq 0$ and for some subset of the input data $Z \subset X$. There, $\Delta f : 2^{\mathcal{Y}} \times 2^{\mathcal{Y}} \to \mathbb{R}_{\geq 0}$ is a function measuring the performance *difference* among two analogous models.

Local interpretations. When an observer agent A is *interpreting* a model M behaviour w.r.t. some input data $Z \subseteq X$, it is actually trying to assign a subjective interpretability value $I_A(R)$ to some representation $R = r(M, Z)$ of choice, aimed at highlighting the behaviour of M w.r.t. the data in Z. There, $r : \mathcal{M} \times 2^{\mathcal{X}} \to \mathcal{R}$ is *representation means*, i.e., a function mapping models into *local* representations w.r.t. a particular subset of the input domain, whereas $\mathcal{R}$ is the set of model representations. For instance, in the case M is a classifier, R may be a graphical representation of (a portion of) the decision boundary/surface for a couple of input features.

There may be more or less interpretable *representations* of a particular model for the same observer A. Furthermore, representations may be either global or local as well, depending on whether they represent the behaviour of the model for the whole input space, or for just a portion of it. For example, consider the case of a plot showing the decision boundary of a neural network classifier. This representation is likely far more interpretable to the human observer than a graph representation showing the network structure, as it synthesise the global behaviour of the network concisely and intuitively. Similarly, saliency maps are an interpretable way to *locally* represent the behaviour of a network w.r.t. some particular input image. So, a way for easing interpretation for a given model behaviour w.r.t. a particular sort of inputs is about looking for the right representation in the eyes of the observer.

Local explanations. Conversely, when an observer A is *explaining* a model M w.r.t. some input data $Z \subseteq X$, it is actually trying to produce a model $M' = E(M, Z)$ through some function $E : \mathcal{M} \times 2^{\mathcal{X}} \to \mathcal{M}$. In this case, we say M' is a *local explanation* for M w.r.t. to Z. We also say that M' is produced through the explanation strategy E.

Furthermore, we define an explanation M' as *admissible* if it has a valid fidelity w.r.t. the original model M and the data in Z—where Z is the same subset of the input data used by the explanation strategy. More precisely, we say M' is δ-admissible in Z w.r.t. M if $\Delta f(M(Z), M'(Z)) < \delta$.

Finally, we define an explanation M' as *clear* for A, in Z, and w.r.t. the original model M, if there exists some representation $R' = r(M', Z)$ which is more interpretable than the original model representation R. More precisely, we say M' is ε-clear for A, in Z, and w.r.t M if $I_A(R') - I_A(R) > \varepsilon$ for some arbitrarily big threshold $\varepsilon > 0$.

Several *explanations* may actually be produced for the same model M. For each explanation, there may be again more or less interpretable *representations*. Of course, explanations are useful if they ease the seek for more interpretable representations. Thus, providing an explanation for a given model behaviour w.r.t. a particular class of inputs is about creating *ad-hoc* metaphors aimed at easing the observer's understanding.

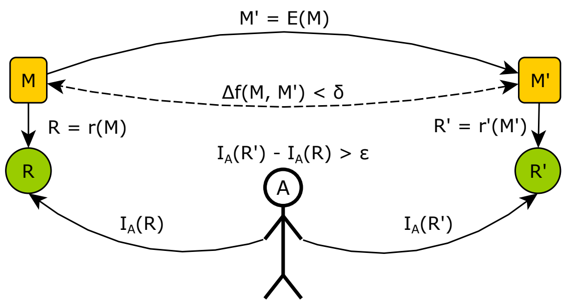


Figure 6.4: Global explanation and interpretation of a model

Global / local explanations. The theoretical framework described so far – which is graphically synthesised in fig. 6.3 – is aimed at modelling *local* interpretations and explanations, that are, the two means an explanator agent may exploit in order to make AI tasks' *outcomes* more understandable in the eyes of some explanee.

Conversely, when the goal is not to understand some model outcome, but the model itself, from a *global* perspective – or, equivalently, when the goal is to understand the model outcome w.r.t the whole set of input data X –, the theoretical framework described so far is simplified as shown in fig. 6.4, where the dependency on the input data is omitted from functions E, Δf, and r. This is possible because we consider the global case as a particular case of the local one, where $Z \equiv X$.

Finally, we remark that the case where a model M is to be understood on a single input-output pair, say x and $y = M(x)$, is simply captured by the aforementioned local model, through the constraint $Z = \{x\}$ and $M(Z) = \{y\}$.

6.2.2 Discussion

Our framework is deliberately abstract in order to capture a number of features we believe to be essential in XAI. First of all, our framework acknowledges – and properly captures – the orthogonality of interpretability w.r.t. explainability. This is quite new, indeed, considering that most authors tend to use the two concepts as if they were equivalent or interchangeable.

Furthermore, our framework explicitly recognises the *subjective* nature of interpretation, as well as the subtly *objective* nature of explanation. Indeed, interpretation is a subjective activity directly related to agents' perception and SoM, whereas explanation is an epistemic, computational action which aims at producing a high-fidelity model. The last step is objective in the sense that it does not depend on the agent's perceptions and SoM, thus being reproducible in principle. Of course, the *effectiveness* of an explanation is again a subjective aspect. Indeed, a clear explanation (for some agent) is a more interpretable variant of some given model—thus, the subjective activity of interpretation is again implicitly involved.

The proposed framework also captures the importance of representations. This is yet another degree of freedom that agents may exploit in their seek for a wider understandability of a given model. While other frameworks consider interpretability as an intrinsic property of AI models, we stress the fact that a given model may be represented in several ways, and each representation may be interpreted differently by different agents. As further discussed in the remainder of this chapter, this is far from being an issue. This subjectivity is deliberate, and it is the starting point of some interesting discussions.

Finally, our framework acknowledges the global/local duality of both explanation and interpretation, thus enabling AI models to be understood either general or with respect to a particular input/output pair.

6.2.3 Practical remarks

The ultimate goal of our framework is to provide a general, flexible, yet minimal framework describing the many aspects concerning AI understandability in the eyes of a *single* agent. We here illustrate several practical issues affecting our framework in practice, and further constraining it.

According to our conceptual framework, a *rational* agent seeking to understand some model M (or make it understandable) may either choose to elaborate on the *interpretation axis* – thus looking for a (better) representation R of M – or it can elaborate on the *explainability axis*—thus producing a novel, high fidelity model M', coming with a representation R' which is more interpretable than the original one (i.e., R).

Notice that, in practice, the nature of the model constrains the set of admissible representations. This means that a rational agent is likely to exploit both the explanation and interpretation axes in the general case—because novel representations may become available through an explanation. We argue and assume that each family of AI models comes with just a few *natural* representations. Because of this practical remark, we expect that, in real-world scenarios, an agent seeking for understandability is likely to "work" on both the interpretation and the explanation axes.

For instance, consider decision trees, which come with a natural representation as a tree of subsequent choices leading to a decision. Conversely, neural networks can either be represented as graphs or as algebraic combinations of tensors. In any case, neural network models are commonly considered less interpretable than other models. In such situation, a rational agent willing to make a neural network more understandable may choose to combine decision trees extraction (explanation) – possibly focusing on methods from the literature [ADT95, CCDO19] – to produce a decision tree whose tree-like structure (representation) could be presented to the human observer to ease their interpretation. The decision-tree like representation is not ordinarily available for neural networks, but it may become available provided that an explanation step is performed.

Another interesting trait of our framework concerns the semantics of clear explanations. The current definition requires explanation strategies to consume a model M with a given representation R and to produce a high-fidelity model M' for which a representation R' exists, which is more interpretable than R. Several semantics may fit this definition. This is deliberate, since different semantics may come with different computational requirements, properties, and guarantees. For instance, one agent may be interested in finding the *best* explanation—that is, the one for which *each* representation is more interpretable than the most interpretable representation of the original model. Similarly, in some cases, it may be sufficient – other than more feasible – to find an *admissible* explanation—that is, a high-fidelity model for which *some* representation exists that is more interpretable than *some* representation of the original model. However, the inspection of the possible semantics and their properties falls outside the scope of this thesis and is going to be considered as a future research direction.

6.2.4 Assessment of the Framework

The abstraction level of the presented framework has also been conceived in order to capture most of the current state of the art. Along this line, this section aims at validating the fitting of the existing contributions w.r.t. the framework presented in section section 6.2.1: if our framework is expressive enough, it should allow most (if not all) existing approaches to be uniformly framed, to be easily understood and compared. To this end, we leverage on the work by Guidotti et al. [GMR$^+$19], where the authors perform a detailed and extensive survey on the state-of-the-art methods for XAI, by categorising the surveyed methods according to an elegant taxonomy. Thus, hereafter, we adopt their taxonomy as a reference for assessing our framework.

The taxonomy proposed by Guidotti et al. essentially discriminates among two main categories of XAI methods. These are the "transparent box design" and the "black-box explanation" categories. While the former category is not further decomposed, the latter comes with three more sub-categories, such as "model explanation", the "outcome explanation", and the "model inspection". Notice that, despite the authors' definition of "explanation" does not precisely match the one proposed in this chapter, we maintained the original categorisation.

The remainder of this section navigates such a taxonomy accordingly, by describing how each (sub-)category – along with the methods therein located – fits our abstract framework.

Model explanation

The mapping of the methods classified as part of the "model explanation" sub-category into our framework is seamless. Hence, it can be defined as follows:

> Let M be a sub-symbolic classifier whose internal functioning representation R is poorly interpretable in the eyes of some explanee A, and let $E(\cdot)$ be some *global* explanation strategy. Then, the model explanation problem consists of computing some *global* explanation $M' = E(M)$ which is δ-admissible and ε-clear w.r.t. to A, for some $\delta, \varepsilon > 0$.

For instance, according to Guidotti et al., possible sub-symbolic classifiers are neural (possibly deep) networks, support vector machines, and random forests. Conversely, explanation strategies may consist of algorithms aimed at *(i)* extracting decision trees/rules out of sub-symbolic predictors and the data they have been trained upon, *(ii)* compute feature importance vectors, *(iii)* detecting saliency masks, *(iv)* detecting partial dependency plots, etc.

In our framework, all the algorithms mentioned above can be described as *explanation strategies*. Such mapping is plausible given their ability to compute an admissible, and possibly more explicit models out of black boxes and the data they have been trained upon. However, it is worth to highlight that the clarity gain produced by such explanation strategies mostly relies on the implicit assumption that their output models come with a natural representation which is intuitively interpretable to the human mind.

Outcome explanation. Methods classified as part of the "outcome explanation" sub-category can be very naturally described in our framework as well. In fact, it can be defined as follows:

> Let M be some sub-symbolic classifier whose internal functioning representation $R = r(M, Z)$ in some subset $Z \subset \mathcal{X}$ of the input domain is poorly interpretable to some explanee A, and let $E(\cdot, \cdot)$ be some *local* explanation strategy. Then, the outcome explanation problem consists of computing some *local* explanation $M' = E(M, Z)$ which is δ-admissible and ε-clear w.r.t. to A, for some $\delta, \varepsilon > 0$

Summarising, while input black boxes may still be classifiers of any sort, explanation, and explanation strategies differ from the "model explanation" case. In particular, explanation strategies in this sub-category may rely on techniques leveraging on attention models, decision trees/rules extraction, or well-established algorithms such as LIME [RSG16], and its extensions—which are essentially aimed at estimating the contribution of every input feature of the input domain to the particular outcome of the black box to be explained.

Notice that the explanation strategies in this category are only required to be admissible and clear in the portion of the input space surrounding the input data under study. Such a portion is implicitly assumed to be relatively small in most cases. Furthermore, the explanation strategy is less constrained than in the global case, as it is not required to produce explanations elsewhere.

Model inspection. Methods classified as part of the "model inspection" sub-category can be naturally defined as follows:

> Let M be a sub-symbolic classifier whose available *global* representation $R = r(M)$ is poorly interpretable to some explanee A, and let $r(\cdot), r'(\cdot)$ be two different representation means. Then, the model inspection problem consists of computing some representation $R' = r'(M)$ such that $I_A(R') > I_A(R)$

Of course, solutions to the model inspection problem vary a lot depending on which specific representation means $r(\cdot)$ is exploited by the explanator, other than the nature of the data the black box is trained upon. Guidotti et al. also provide a nice overview of the several sorts of representations means which may be useful to tackle the model inspection problem, like, for instance, sensitivity analysis, partial dependency plots, activation maximization images, tree visualisation, etc.

It is worth pointing out the capability of our framework to reveal the actual nature of the inspection problem. Indeed, it clearly shows how this is the first problem among the ones presented so far, which only relies on the interpretation axis alone to provide understandability.

Transparent box design. Finally, methods classified as part of the "transparent box design" sub-category can be naturally defined as follows:

> Let $X \subseteq \mathcal{X}$ be a dataset from some input domain $\mathcal{X}$, let $r(\cdot)$ be a representation means, and let A be the explanee agent. Then the transparent box design problem consists of computing a classifier M for which a global representation $R = r(M, X)$ exists such that $I_A(R) > 1 - \delta$, for some $\delta > 0$

Although very simple, the transparent-box design is of paramount importance in XAI systems as it is the basic brick of most general explanation strategies. Indeed, it may be implicit in the functioning of some explanation strategy E to be adopted in some other model or outcome explanation problem.

For instance, consider the case of a local explanation strategy $E(M, X) \mapsto M'$. In the general case, to compute M', it relies on some input data X and the internal of the to-be-explained model M. However, there may be cases where the actual internal of M are not considered by the particular logic adopted by E. Instead, in such cases, E may only rely on X and the outcomes of M, which are $Y = M(X)$. In this case, the explanation strategy E is said *pedagogical*—whereas in the general case it is said *decompositional* (cf. [ADT95]).

In other words, as made evident by our framework, the pedagogical methods exploited to deal with the model or outcome explanation problems must internally solve the transparent box design problem, as they must build an interpretable model out of some sampled data-set and nothing more.

6.3 Symbolic Knowledge Extraction

This section contains contributions from the following works of ours: [CCOC19, CCO20, CCDO19]

Many strategies can be exploited to pursue the purpose of explainability [GMR+19]. Some authors suggest for instance to *only* rely on *interpretable* algorithms [Rud19] – such as generalised linear models, decision trees, etc. – to construct data-driven solutions that are explainable by construction. However, this may hinder predictive performance in the general case, as it essentially cuts off most effective algorithms—e.g., ANN. Another strategy consists of deriving *post-hoc* explanations [KFQK21], aimed at reverse-engineering the inner operation of a BB so as to make it explicit. In this way, data scientists can keep using prediction-effective predictors such as NN, while still attaining high predictive performance. The focus of this section is on the latter strategy.

Symbolic knowledge extraction (SKE) [dGBG01] is among the most promising means to derive *post-hoc* explanations for sub-symbolic predictors. Roughly speaking, the main idea behind SKE is to enable the construction of a *symbolic* surrogate model mimicking the behaviour of a given predictor. There, symbols may consist of intelligible knowledge, such as flat lists or hierarchical trees of *rules*. Such rules can then be exploited to either derive predictions or to better understand the behaviour of the original predictor.

SKE has been applied, for instance, to credit-risk evaluation [BSMV03, BSDL+01, SNS+06], healthcare – i.e., to make early breast cancer prognosis predictions [FSM+07] and to help the diagnosis and discrimination among hepatobiliary disorders [HSY00] or other diseases and dysfunctions [BP97, Bol00] –, credit card screening [SBM11], intrusion detection systems [HSS03], and keyword extraction [ALS12].

6.3.1 State of the art

Here we discuss the main approaches for extracting symbolic knowledge out of sub-symbolic predictors.

The main underlying assumption behind most works in this category is that, once a sub-symbolic system has been trained over large amounts of data reaching some good predictive performance, then it must have attained a distributed representation of the knowledge contained in the data. Even though unintelligible to human beings, the distributed representation is still somehow buried in the internals of that sub-symbolic systems. Assuming this is the case, then a knowledge extraction technique is a means for making the distributed representation explicit and intelligible.

It is worth to mention that the idea of extracting decision rules or trees from sub-symbolic predictors is not new: it has been introduced several times, in many forms, and with different names and methods, since the late 80s. In fact, generally speaking, systems supporting symbolic knowledge extraction have a number of appealing features. In particular, they support a full exploitation of sub-symbolic techniques, which are the best choice when information must be mined from large amounts of data, and are usually better suited in terms of precision, robustness, and predictive performance. However, thanks to the knowledge extracted, those systems retain desirable XAI-related properties which would otherwise be lost.

Knowledge extraction techniques can be described and discriminated according to a number of orthogonal dimensions, including:

1. the structure of the symbolic knowledge they extract (e.g., decision rules, decision trees, etc)

2. the type of constraints they exploit for decision-making (e.g., linear constraints, M-of-N rules, etc)

3. the sort of sub-symbolic predictor(s) they can deal with (e.g., neural networks, support vector machines, etc)

In the reminder of this section we partition the surveyed works according to dimensions 1, and 3, then for each approach we discuss the sort of the constraints exploited. In particular, in the same way as other impactful surveys on the topic [GMR$^+$19, ADT95], we distinguish between techniques extracting *rule* lists and techniques extracting decision *trees*. Then, we further distinguish between pedagogical and decompositional approaches. In doing so we borrow the terminology from [ADT95], where *pedagogical* techniques are those capable of extracting symbolic knowledge from any sort of sub-symbolic predictor – as they do *not* exploit any internal detail of the predictor under study to perform the extraction –, whereas *decompositional* techniques are those only capable of extracting symbolic knowledge from some particular sort of sub-symbolic predictor (e.g., neural networks, in most cases)—as they perform the extraction by looking at the internals of the predictor at hand.

Rules extraction

Here we focus on methods for extracting *flat* list of rules in the form

$$\begin{array}{c}\textbf{if } \mathit{condition}_1 \textbf{ then } \mathit{outcome}_1 \\ \textbf{else if } \mathit{condition}_2 \textbf{ then } \mathit{outcome}_2 \\ \vdots \\ \textbf{else } \mathit{outcome}_n\end{array}$$

out of sub-symbolic predictors, where each *condition* is can be a conjunction or disjunction of *(i)* boolean predicates, *(ii)* linear constraints, or *(iii)* M-of-N rules over the attributes of the data used to train the sub-symbolic predictor.

We categorise the surveyed techniques for rule extraction depending on whether they are decompositional or pedagogical; then we provide some details for each technique; finally, we analyse them from the XAI perspective in an aggregate manner, given the huge similarity characterising the surveyed techniques from the XAI perspective.

Pedagogical approaches. We identified three main pedagogical approaches for rule extraction:

- the method from Saito et al. (1988) [SN88]
- RxREN (2012) [AK12]
- ALPA (2015) [dFM15]

In particular, [SN88] extracts M-of-N rules out of any black-box classifier, regardless of whether it is a neural network or not. Apparently, however, this method does not support regression, and it only supports categorical attributes as conditions in the extracted rules. In spite of its limitations, this work has been proven to be effective in expert systems for diagnosis support.

On the contrary, [AK12] and [dFM15] extract if-then-else rules out of arbitrary classifiers. In particular, RxREN supports datasets with mixed mode attributes (i.e., either categorical or numerical). The algorithm is based on a reverse-engineering algorithm that essentially discards insignificant attributes and discovers the variation range of input attribute for each possible outcome of the classification. For this reason, the rules extracted by RxREN are composed by linear constraints. Conversely, the ALPA rule extraction technique is the first that is applicable to any black-box model with no limitations on the nature of constraints.

It is worth remarking that pedagogical approaches are not based on the structure of the network, therefore they also work with other sub-symbolic models—even though oldest papers tend to mention neural networks more than other sorts of predictors.

Finally, it is worth to be mentioned that pedagogical extraction algorithms can essentially be described as oracle-based algorithms. In fact, in most cases the extraction algorithm works by querying the black box (which is therefore considered as an oracle), and by using the corresponding responses to build the rule list. This behaviour is repeated until the set of rules given by the white-box model converges to that of the black box. In other words, the extraction procedure terminates when the rule set as whole reaches a high *fidelity* w.r.t. the original black box.

Decompositional approaches. We identify some main decompositional approaches for rule extraction:

- RuleNet (1992) [MMS92]
- MofN (1992) [TS92]
- the method from Giles et al. (1993) [GO93]
- KT (1994) [Fu94]
- VI-Analysis (1995) [Thr95]
- RX (1997) [Set97]
- the method from Núñez et al. (2008) [NAC08]

Generally speaking, most approaches here explicitly target a particular sort of sub-symbolic predictor. In particular, all approaches except [NAC08] target neural networks, whereas [NAC08] target support vector machines (SVM).

Some approaches [MMS92, TS92, GO93] exploit some strict assumptions that limit the kind of neural networks they can manage, thus reducing their generality. For instance, the RuleNet technique described in [MMS92] can only handle neural networks aimed at computing endomorphisms on n-sized strings of characters, and it aims at making explicit the condition-action rules exploited by such sorts of networks. At the same time, the MofN technique [TS92] can only handle neural networks attained via the KBANN algorithm described above. As suggested by its name, this method extracts M-of-N-like rules. Finally, the method proposed in [GO93] focuses on neural networks trained to act as recognisers for regular languages, and it is capable of extracting rules in the form of finite state automata for parsing these languages.

Other approaches – e.g., [Fu94, Thr95, Set97] – target general purpose neural networks. Briefly speaking, they compile networks into sets of rules with equivalent structure. There, each processing unit (neuron) is mapped into a separate rule – or a small set of rules –, and the in-going connections are interpreted as preconditions to that rule. The particular shape of preconditions – e.g., linear constraints, M-of-N constraints, etc. –, is then inferred by taking into account the weights of a neuron in-going connections, and its activation function. For instance, the KT algorithm [Fu94] is capable of learning if-then-else rules with linear constraints out of general neural-network classifier. Similarly, the VI-Analysis [Thr95] and RX [Set97] algorithms perform the same task via different procedures.

Finally, a different and noteworthy approach is described in [NAC08]. There, the authors propose a method for extracting if-then-else rules with linear constraints out of SVM classifiers.

XAI Perspective. Generally speaking, rule extraction techniques provide *post-hoc* explainability via model simplification. In fact, all the surveyed extraction procedures aims at creating a list of rule having a high-fidelity w.r.t. the source black-box predictor. This rule list can then be considered a symbolic, intelligible explanation of the source predictor. Accordingly, we argue that all these techniques may contribute to the pursuit of XAI goals as: trustworthiness, causality, transferability, informativeness, confidence, and possibly fairness. In fact, by making the inner operation of a black-box predictor explicit and intelligible, these techniques may increase the trustworthiness and confidence of intelligent systems. Furthermore, by providing an overview of the all the possible context-decision situation an intelligent system may face, and by making it possible to inspect which particular rule lead to a particular decision, rule extraction techniques may provide informativeness and causality. Moreover, the symbolic knowledge extracted can be translated into several forms, possibly making it compliant with symbolic intelligent systems. This of course provides for transferability. Finally, rule extraction techniques may help with fairness as well, by highlighting the biases possibly learned by sub-symbolic predictors.

It is worth to mention, however, that rule extraction technique are not the silver bullet of XAI. Issues related to accuracy, fidelity, and consistency, may easily arise in this kind of approaches, because the extracted rule list may not perfectly reflect insights of the original one. We argue that this is essentially unavoidable: the extracted rule lists are essentially approximated models, which are attained by removing (i.e. loosing) information from the source black-box. Moreover, interpretability of the extracted rule list may easily deteriorate as the amount of rules (or the amount of predicates per rule) increases—a situation which may easily arise as the complexity or the dimensionality of the black-box become non-trivial. Finally, it is worth to be noted that virtually all rule extraction techniques only focus on black-box predictors acting as classifiers. Not so much attention has been devoted by the academic community to the extraction of rules out of sub-symbolic regressors, as well as black boxes aimed at performing unsupervised learning tasks.

As a side note concerning SVM-based rule extraction techniques, it is worth to be mentioned that, although they have been known to produce classifiers that are easily comprehensible, they often approximate secondary models of worse accuracy [BB10]. Moreover, even though these models may be reasonably understandable from an expert perspective, they still lack the simplicity and familiarity to an individual user that often intelligent systems have to provide, as in the case of recommendation.

Decision trees extraction

Here we focus on methods for extracting *hierarchical* decision tree out of sub-symbolic predictors.

Generally speaking, extracted decision trees are ordinary decision trees whose nodes are represented by rules consisting of a conjunction or disjunction of *(i)* boolean predicates, *(ii)* linear constraints, or *(iii)* M-of-N rules over the attributes of the data used to train the sub-symbolic predictor, similarly to the aforementioned decision rules. In other words, the main difference with decision rules lays in the hierarchical nature of decision choices.

Given the small amount of techniques for decision tree extraction surveyed in this section, we do not split our discussion any further to distinguish between pedagogical or decompositional approaches. Rather we provide this information as part of the detailed description of each method, provided below. We provide a joint discussion of decision tree extraction methods from the XAI perspective, at the end of the paragraph.

Surveyed methods. We identify three main approaches for decision tree extraction:

- TREPAN (1996, pedagogical) [CS95]
- the method by Krishnan et al. (1999, pedagogical) [KSB99]
- the method by Schetinin et al. (2007, decompositional: random-forest-specific) [SFP$^+$07]

TREPAN [CS95] is a pedagogical tree extraction algorithm that extracts decision trees from sub-symbolic models. TREPAN grows a tree through recursive partitioning, using a best-first expansion strategy, towards M-of-N-like, tree-structured rules. The black box model – be it a neural network, a support vector machine, or any other model that can be used for classification – is used as an oracle to answer questions of class belongingness on artificially-generated data points. It also exploits the active learning process to additionally generate data points according to network constraints.

Along the same line, [KSB99] proposes decision tree extraction from neural networks. Unlike Trepan, however, the internal structure of the neural network is taken into consideration in the process of decision tree construction. Furthermore, while Trepan leverages on a restricted form of active learning, the method proposed by [KSB99] leverages on a genetic algorithm. Finally, it is worth mentioning that the latter algorithm supports the extraction of trees of a given size. In other words, the size of extracted tree can be tuned.

Finally, [SFP+07] proposes an approach for the probabilistic interpretation of Bayesian decision trees ensembles (a.k.a. random forests) as a single decision tree. Classification confidence for each tree in the forest is calculated by exploiting training data: the decision tree covering the maximum number of correct training examples is selected, keeping the amount of classification errors in the remaining examples minimal. Unlike the previous ones, this method of explanation does not extend the input data set with random additional data and cannot be generalised to other types of sub-symbolic black boxes.

XAI Perspective. From the point of view of XAI, decision tree extraction methods are quite similar to rule extraction ones, thus similar concerns fit their case. Accordingly, we argue that decision tree extraction techniques provide *post-hoc* explainability via model simplification, and help in pursuing XAI goals such as trustworthiness, causality, transferability, informativeness, confidence, and fairness.

In spite of the many similarities with rule extraction techniques, a remarkable peculiarity of decision trees extractors is worth to be mentioned: as hierarchical models, they are less prone to interpretability issues when the complexity or dimensionality of the source predictor grows.

Other critical aspects remain however unresolved w.r.t. rule extraction techniques. These include issues arising from the potential lack of fidelity, as well as the concentration of practically all the decision tree extraction techniques on the sole case of classification tasks—leaving others kinds of tasks in machine learning essentially uncovered.

6.3.2 A practical framework for MAS

This subsection contains contributions from the following works of ours: [SCCO21]

In the reminder of this thesis, we commit to computational logics as the preferred means for explainability. In particular, we sketch a general framework for XAI, reifying the abstract framework from section 6.2.1 via SKE.

The basic idea is that SKE can be used to draw explanations in the form logic knowledge, which can then be further symbolically manipulated to ease the interpretation of the users. Explanations, in this case, consist of *surrogate* predictors

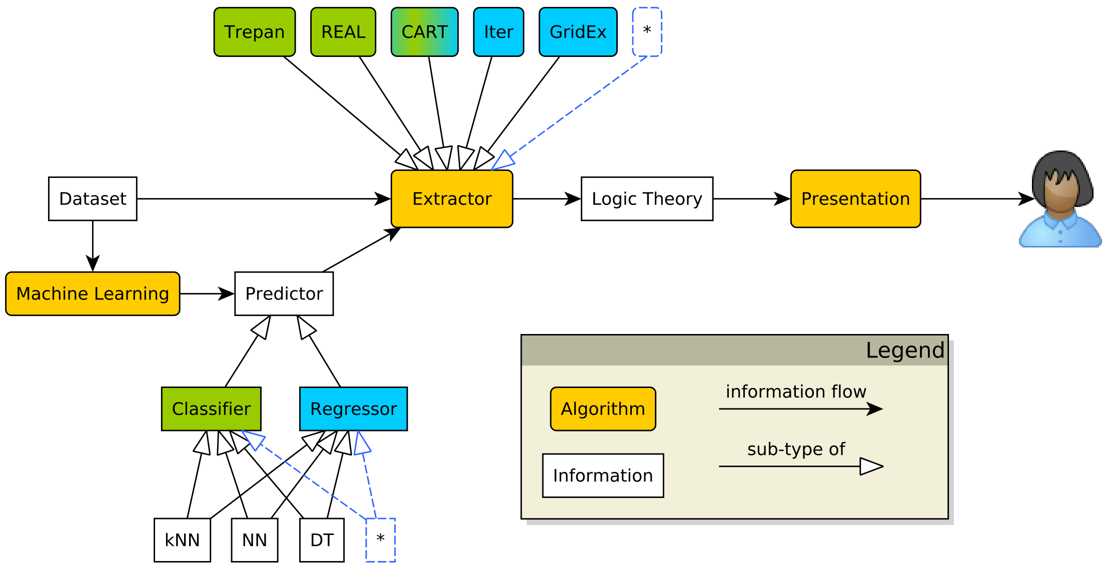


Figure 6.5: A practical framework for XAI based on SKE

trained to mimic the ones to be explained, as closely as possible. Such surrogates consist of logic knowledge bases, intensively representing the internals of some sub-symbolic ML predictor via interpretable formulæ. In fact, given a trained predictor and a knowledge-extraction procedure applicable to it, the extracted rules/trees act as *explanations* for that predictor – or as a basis to build some –, provided that they retain high *fidelity* w.r.t. both the original predictor and the data used to train it.

Framework design. In particular, as depicted in fig. 6.5, a pivotal role in the design of our practical framework is played by the notion of *extractor*, defining the general contract of any knowledge extraction procedure. More precisely, any algorithm accepting a *ML-trained* predictor – either a classifier or a regressor – as input, and producing logic rules as output can act as an extractor. Thus, the many algorithms described in section 6.3.1 are well suited to act as extractors.

To perform their job, extractors may require additional information about the data the input predictor has been trained upon. In the general case, such information consists of the data set itself and its schema—i.e., a formal description of the names and the data types of all features characterising the data set itself. Data sets are required to let extraction procedures inspect BB behaviour – and therefore build the corresponding output rules –, whereas schemas are required to *(i)* let the extraction procedure take informed decisions on the basis of the feature *types*, *(ii)* let the extracted knowledge be clearer by referring to the feature *names*. For all these reasons, extractors expect a data set and its schema metadata to be provided in input as well.

We stress the logic nature of the extractors' outputs. These may consist of *knowledge bases* containing logic rules expressed via some logic formalism of choice—e.g. FOL, Horn clauses, or ProbLog. These rules should *intensively* represent the general behaviour of the original predictor, in a symbolic way. Furthermore, the actual behaviour of the original predictor on a particular input should be reproducible via inference. Thus, the whole knowledge base acts as a *global* explanation for the original predictor, whereas the proof tree aimed at inferring a particular output acts as a *local* explanation for that particular prediction.

Of course, logic rules may still be poorly interpretable to inexpert human agents. The formalism may for instance be obscure for the human, the rules may be too many, their arity may be too large, or their bodies may involve too many literals. For all these reasons, the framework assumes one further processing step, aimed at representing explanations in the most adequate way for the human agent consuming them. While the possible representations are manifold, the key point here is that the extraction of logic knowledge out of sub-symbolic predictors is not the ultimate step of explanation but rather an intermediate one, enabling the *inspection* of the inner operation of some black-box predictor.

Consider for instance the case of a sub-symbolic predictor $\mathcal{P}$ (say, a neural network) trained on the well-known Iris data-set to classify iris flowers as one of the `setosa`, `versicolor`, or `virginica` types, by only looking at a flower's sepal width (SW) and height (SH), and petal width (PW) and (PH). Suppose that the following logic theory $\mathcal{T}$ is extracted by means of some extraction procedure:

$$\begin{array}{rcl}
iris(SL, SW, PL, PW, \texttt{setosa}) & \leftarrow & PW \leq 0.65. \\
iris(SL, SW, PL, PW, \texttt{versicolor}) & \leftarrow & PW \in \,]0.65, 1.64]. \\
iris(SL, SW, PL, PW, \texttt{virginica}) & \leftarrow & PW > 1.64.
\end{array}$$

In the hypothesis that $\mathcal{T}$ has an high fidelity w.r.t. $\mathcal{P}$, the as a whole theory provides useful insights about how it works. For instance, it demonstrates how the "petal length" feature plays a central role in the strategy followed by $\mathcal{P}$ to classify iris flowers. Furthermore, given a particular query such as $iris(4.9,\ 2.4,\ 3.3,\ 1,\ C)$, and the proof tree corresponding to a resolution attempt of that query w.r.t $\mathcal{T}$, each query-to-solution path is highly informative:

$$\frac{\dfrac{\perp \quad (a)}{C \mapsto \texttt{setosa} \quad \cancel{1 \leq 0.65}} \qquad \dfrac{\top \quad (b)}{C \mapsto \texttt{versicolor} \quad 1 \in \,]0.65, 1.64]} \qquad \dfrac{\perp \quad (c)}{C \mapsto \texttt{virginica} \quad \cancel{1 > 1.64}}}{iris(4.9,\ 2.4,\ 3.3,\ 1,\ C)}$$

There, path (a) explains why the iris flower above is *not* of type `setosa` – i.e., because its petal width (1) is greater than 0.65 –, path (b) explains why it is of type `versicolor` – i.e., because its petal width is in the $0.65 \div 1.64$ range –, and path (c) explains why the iris flower is *not* of type `virginica`—totally analogous to the `setosa` case.

Finally, more user-friendly representations could be generated for both rules and proof tree paths. The topic is further discussed in [CCDO19].

Why logic. The exploitation of computational logic as the basic means to draw explanations is strategical under a number of perspectives. First, we argue that symbolic representations (e.g., the language of FOL formulæ), may act as a *lingua franca* for knowledge representation and exchange among heterogeneous intelligent agents—there including humans. Second, we believe that the adoption of symbolic AI to be an enabling choice for the full exploitation of MAS. Finally, this would enable XAI technologies to benefit form the wide gamma of results, methods, algorithms, and toolkits developed under the umbrella of symbolic AI.

In particular, the potential of logic-based models and their extensions is mainly due to their *declarativeness* and *explicit knowledge representation* – enabling knowledge sharing at an adequate level of abstraction – modularity, and separation of concerns [OP11]—which are especially valuable in open (and possibly distributed) systems composed by several intelligent agents. The interested readers may consider reading [CCOC19, CCN^{+}21] for a more detailed description of the expected benefits provided by a logic based approach to XAI.

Part II

How

Chapter 7

The Role of Logic Based Technologies

This chapter contains contributions from the following works of ours: [CCDO20]

Artificial intelligence (AI) is getting ever growing attention from both the academia and the industry, in terms of resources, economic impact, available technologies and widespread adoption in virtually any application area. In fact, more and more industries are adopting and applying state-of-the-art AI techniques, to actively pursue challenging business objectives.

Such a general interest, and technology adoption, has been favoured by two main ingredients: *(i)* the development of advanced technologies even at the micro scale, and *(ii)* the availability of large amounts of data in the environment around us to learn from. These ingredients have boosted *sub-symbolic* AI techniques, such as machine learning (ML), – there including deep learning and neural networks – aimed at exploiting big data to make predictions and take autonomous decisions—in contrast to the more long-established *symbolic* techniques, based on the formal representation of knowledge and its elaboration via explicit reasoning rules.

The increasing role of intelligent systems in human society, however, raises unprecedented issues about the need to *explain* the behaviour, or the result of, intelligent systems—in the sense of being capable of *motivating* their decision and make the underlying decision process *understandable* by human beings: this is where sub-symbolic techniques, despite their efficiency, fall short. This is especially relevant when AI is exploited in the context of human organisations meant at providing public services—such as, e.g., health care / diagnostic systems or legal advice. There is therefore an emerging need to reconcile and synthesise symbolic and sub-symbolic techniques, exploiting the first to *explain* the latter—the scope of *eXplainable Artificial Intelligence* (XAI) [Gun16].

So, just when AI's general focus is on sub-symbolic techniques, symbolic approaches are re-emerging as the means to bring AI closer to the human understanding, helping humans to overcome fears and ethical issues by providing explainability, observability, interpretability, responsibility, and trustability. In this context *logic-based* approaches, despite their age, are finding a new youth, for a number of reasons—first of all, because of their closer relation to the human cognitive process. Moreover, their role in (not just computer) science is well understood – think for instance of the formal study of programs and semantics in computational models, computational logic, inference as computation, logic programming, and automatic theorem proving [Gal85, BM88]. Last but not least, logic-based approaches have long been at the centre of many successful agent-based models and technologies: indeed, agents reason through logic, and plan and coordinate through logical processes [Lev84, ODN95, BBD$^+$06]. Overall, from the XAI spark, wider perspectives are raising also beyond the symbolic/sub-symbolic dichotomy, yet all sharing a logic-based root.

Accordingly, in this chapter we focus on the role that logic technologies have played over the years and are going to play in the forthcoming AI landscape—in particular, for the engineering of intelligent systems. We leverage on that analysis to identify the most promising research perspectives and the open issues characterising the current state of the art.

7.1 Logic-based AI: Application Areas

The above techniques have been applied in a variety of fields: fig. 7.1 summarises both the main application areas and their interconnections. For each application area, in the following we *(i)* introduce what the area is about, *(ii)* outline the main sub-categories (if any), *(iii)* discuss the role of logic in each sub-category as well as *(iv)* its benefits and limits, and *(v)* present the main actual applications.

In order to make the comparison actually effective, table 7.1 and table 7.2 summarise our findings from two different viewpoints. The first outlines the strengths of the diverse techniques per application area, that is, where they provide the strongest contribution; the second puts each technique in relation with the different market segments, providing appropriate references to the literature where each technique is used.

7.1.1 AI Foundations

Formalisation & Verification of Computational Systems

Formalisation and verification of computational systems refer to a collection of techniques for the automatic analysis of reactive systems—in particular, safety-critical systems, where subtle design errors can easily elude conventional simulation and testing techniques.

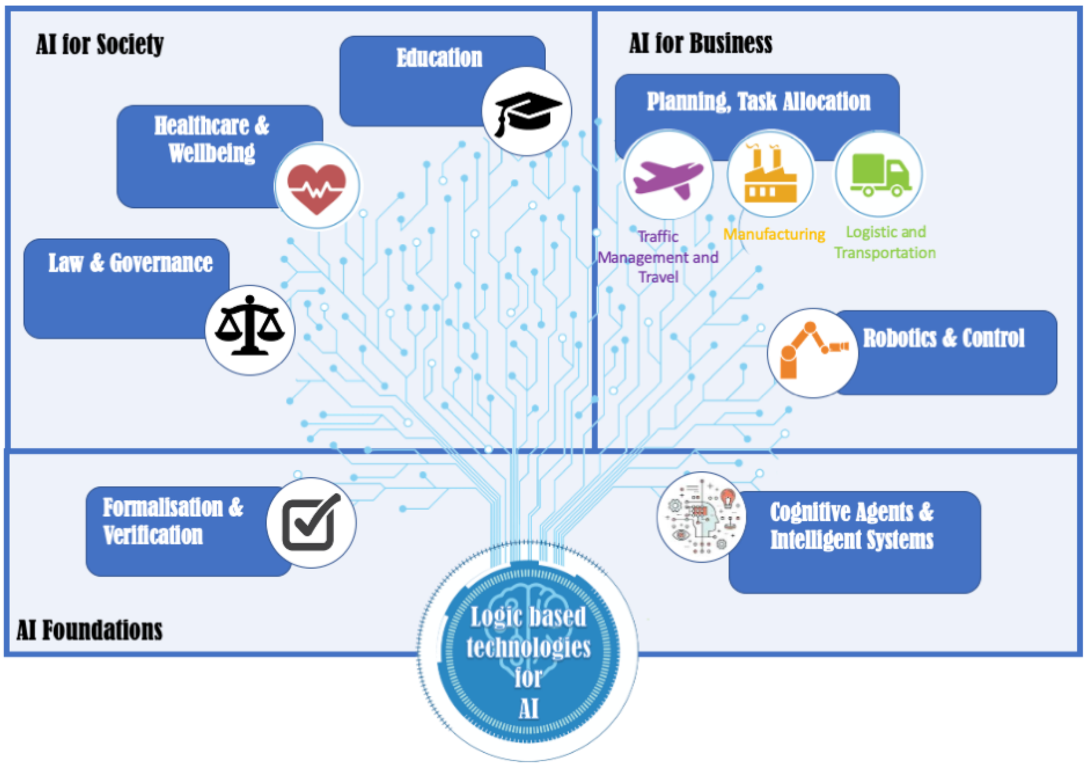

AI for Society
Education
Healthcare & Wellbeing
Law & Governance
AI for Business
Planning, Task Allocation
Traffic Management and Travel
Manufacturing
Logistic and Transportation
Robotics & Control
Formalisation & Verification
Cognitive Agents & Intelligent Systems
AI Foundations
Logic based technologies for AI

Figure 7.1: Logic-based technologies application areas with respect to main AI categories—namely, AI Foundations, AI for Society, and AI for Business. Intentionally, the picture only illustrates the AI areas that are closely related to logic

	FOL	DL	BDI	TL	FL	PL	DR	CLP
Formalisation & Verification				✓		✓		
Cognitive Agents	✓	✓	✓	✓	✓		✓	
Health-care & Wellbeing	✓	✓				✓		
Law & Governance	✓						✓	
Education	✓	✓			✓			
Planning & Task Allocation	✓	✓					✓	✓
Robotics & Control		✓		✓	✓			✓

Table 7.1: Sorts of logic per application area. Acronym and abbreviation key: FOL: First-Order Logic; DL: Description Logic; BDI: Belief Desire Intention (Logic); TL: Temporal Logic; FL: Fuzzy Logic; PL: Probabilistic Logic; DR: Defeasible Reasoning; CLP: Constraint Logic Programming.

	FOL	DL	BDI	TL	FL	PL	DR	CLP
Aerospace					[CCS+16, OSS15]			[JM94]
Analytics	[CM12]		[LKR+16]					
Bioinformatics	[CM12]	[YKZ03, Hor05]			[TN06, DR+08b]	[KNP02]		[JM94]
BPM	[CM12]	[YKZ03]						
Constructions								[JM94]
Critical systems				[PVB+13, Wu17]				
CPS		[YKZ03]	[LKR+16, Win05]	[Hol97]	[AES17, MTCB17]	[KNP02]		
Cybersecurity					[Goz12, LMVW11]	[KNP02]		
Databases		[Hor05]			[KYZ00, KZZ89]			
Decision support					[LL09, ADBGDP04]		[GCS13]	
Energy		[Len95]	[LKR+16]	[Hol97]	[SIS15]	[KNP02]		[JM94]
Finance		[Len95]			[Boj07, GL05]			[JM94]
Government & Legal							[Pra13]	
Hardware		[Len95]			[CDGF+95, BZ98]			[JM94]
Healthcare			[LKR+16, CMR+19]	[Hol97]	[AvKLM01, YH12]			
Information retrieval		[Hor05]			[GSS15, HCCL05]			
Logistic	[CM12]	[Len95]	[LKR+16, Win05]	[Hol97]				[JM94]
Manufacturing			[LKR+16, Win05]	[Hol97]	[APG+11, LL05]			
Mechanics								[JM94]
Mobile applications				[KYA+16, MMN18]				
Railways				[Wu17]	[SA11, SKT04]			
Telecommunications	[CM12]				[GRSC98, CFPP96]	[KNP02]		[JM94]
Transports			[LKR+16]		[SSS12, QNO06]			
Web services		[YKZ03, Hor05]		[DMRT06, AKD+10]	[TT08, CYLT05]			

Table 7.2: Sorts of logic per market segment. Acronym and abbreviation key: FOL: First-Order Logic; DL: Description Logic; BDI: Belief Desire Intention (Logic); TL: Temporal Logic; PL: Probabilistic Logic; DR: Defeasible Reasoning; CLP: Constraint Logic Programming.

The main logic-based technology exploited in this field is model checking [CGP01]—nowadays a standard procedure for quality assurance both because of its cost-effectiveness and of the ease of integration with more conventional design methods. The model checker input is a description of the system to be analysed and a number of properties that are supposed to hold: logic is used both to formalise the system description – states, transitions, model description, and specifications to be verified – and to express the behavioural aspects, capturing the key properties of information flow. Accordingly, such descriptions are often expressed in temporal and probabilistic logic (and their extension/variations).

Model checking can provide a significant increase in the level of confidence of a system, enabling system verification a-priori, a-posteriori, and – what is most relevant nowadays – at runtime. On the other hand, any validation is, by definition, only as good as the system model itself: so, the validation result strongly depends on the precision of the input model. In addition, model checking can turn out to be unsuitable for data intensive applications, as it increases the number communications.

Cognitive Agents & Intelligent Systems

Cognitive architectures are design methodologies, i.e., collections of knowledge and strategies applied to the problem of creating *situated intelligence*. Here, the main technologies borrow from *multi-agent systems* (MAS), since the cognitive architecture can be considered as the brain of an agent reasoning to solve problems, achieving goals and taking decisions.

Generally speaking, cognitive agents in intelligent MAS straightforwardly exploit the logic-based models and technologies for the rational process, knowledge representation, expressive communication, and effective coordination. Developing an agent means to set up a deduction process: each agent is encoded as a logic theory, and selecting an action means to perform a deduction that reduces the problem to a solution, as in theorem proving. Logic can also be used to represent the agent's surrounding *environment* and the *society* of agents—that is, overall, two of the three key aspects when it comes model the structure and dynamics of non-trivial MAS [Omi01].

Technologies reflect the above context: many are related to agent programming and reasoning, others to agent reliability and verification. Many others focus on the societal aspect of cognitive architectures, by interpreting society as the ensemble where the collective behaviour of the MAS are *coordinated* towards the achievement of the global system goals. Along this line, *coordination models* glue agents together by governing agent interaction, paving the way towards social intelligence: after the seminal work of Shared Prolog [Cia94], notable examples are TuCSoN [OZ99], ReSpecT [OD01], and AORTA [JDV14].

Within an agent society, agents can enter into argumentation processes to reach agreements and dynamically adapt to changes: so, disputes and conflicts need to be managed in order to achieve a common agreement and establish the winner argument. A number of technologies exist for solving reasoning tasks on the abstract argumentation frameworks [Dun95]. Since problems of this kind are intractable, efficient algorithms and solvers are needed. As discussed in [GLMW18], most solvers are based on logic-based programming or logic frameworks including ASP and SAT-solvers.

Specific technologies exist for dealing with the *environment* abstraction of cognitive architectures, mostly in the *coordination* area. There, coordination/interaction artefacts work as runtime abstractions which encapsulate and support coordination/interaction services, usable as building blocks for designing and governing coordination, collaboration, competition services inside heterogeneous MASs. *Description spaces with fuzziness* [NOV11] and *semantic tuple centres* [NVP10] can be read as technologies for *situated interaction & coordination*, emphasising the *situated* aspect of interaction, i.e., the environment-related aspect. In-between lies LPaaS (Logic Programming as a Service), a framework aimed at supporting the distribution of logic knowledge in the environment. There, artefacts work as knowledge repositories (in the form of environment structure and properties) while also embedding the reasoning process (enabled and constrained by the knowledge they embody).

When agents are immersed in a Knowledge-Intensive Environment (KIE), the cognition process goes beyond that of the individual agent, and distributed cognition processes may take place, promoting the idea of *intelligent environment* [HHK00]. In such a way, the environment concept is extended beyond situated action—which, by the way, motivates the inclusion of the semantic web within the macro-area of environmental abstractions.

Moving from [Hen01], the "Agents *in* the semantic web" sub-category lists JADL, AgentOWL, and EMERALD, which exploit semantic web technologies to inter-operate.

Hybrid cognitive architectures have recently gained attention to combine symbolic and sub-symbolic (emergent) approaches. Examples are ACT-R [AL03] – based on the modelling of human behaviour –, CLARION [Sun05] and LIDA [FMDS14].

In spite of the simplicity and elegance of the logical semantics in logic-based architectures, some issues do exist. The *transduction* problem [BC04] has to do with the difficulty of accurately translating the model into a symbolic representation, especially in a complex environment. One more difficulty comes from suitably representing information in such a symbolic form that agents can reason about, with and in a time-constrained environment. Finally, the transformation of percepts may not be accurate enough to describe the environment itself, due to sensor faults, reasoning errors, etc.

7.1.2 AI for Society

Healthcare & Well-being

In the healthcare domain, AI typically takes the form of complex algorithms and software systems to emulate human cognition in the analysis of complicated medical data, approximating conclusions without direct human input. The primary aim here is to analyse relationships between prevention or treatment techniques and patient outcomes. AI programs have been developed and applied to practices such as diagnosis processes, treatment protocol development, drug development, personalised medicine, and patient monitoring and care.

In this field, logic is exploited to represent knowledge in a human understandable way, and reason on it via properly-formalised rules—in particular, decision support (symbolic) rules, obtained from domain experts and/or decision models induced from data.

At the same time, symbolic logic scales does not scale easily: knowledge engineers need to extract the logic by interviewing or observing human experts. On the other hand, sub-symbolic techniques such as supervised deep learning scale more easily, but are subject to bias in the training data—and, of course, their outcome cannot be explained.

Here again, the semantic web provides a technical framework for the formal semantic modelling – i.e. interpretation, abstraction, axiomatisation, and annotation – of healthcare knowledge in terms of classes, properties, relations and axioms [BLHL01]. The semantic web framework for healthcare systems provide notable features: *(i)* semantic modelling of the procedural and declarative healthcare knowledge as ontologies, hence a semantically rich and executable knowledge representation formalism; *(ii)* annotation – typically via RDF (Resource Description Framework) – of healthcare knowledge artefacts guided by the ontological model of the knowledge artefact, so as to characterise the main concepts and relations within the artefact; *(iii)* representation of different patient data sources in a semantically-enriched formalism, that helps to integrate heterogeneous data sources by establishing semantic similarity between data elements; *(iv)* semantic interoperability between multiple ontologies, using ontology alignment and mediation methods to dynamically synthesise / shape multiple knowledge resources so as to address all the facets of the specific healthcare problem; *(v)* specification of the decision-making logic in terms of symbolic rules, which can be executed using proof engines to infer suitable recommendations/actions; and *(vi)* provision of a justification trace of the inferred recommendations, so as to let users understand the rationale of the recommended interventions [Abi08].

Among the healthcare systems based on reasoning, CARA (Context Aware Real-time Assistant) [YH12] aims at providing personalised healthcare services for chronic patients in a timely manner, adapting the healthcare technology so that it fits in both with the normal activities of the elderly and with the working practices of the caregivers. Based on a fuzzy-logic context model and a related context-aware reasoning middleware, CARA provides context-aware data fusion and representation, as well as inference mechanisms that support remote patient monitoring and caregiver notification.

Law & Governance

Due to its wide potential impact on society and economy, *AI & law* is one of today's most relevant research areas. It consists of an interdisciplinary effort combining methods and results from several sources, from deontic logic, norms and agent-based simulation to game theory and norms, normative agents, norms and organisation, norms and trust, norms and argumentation.

Contributions in the field of AI & law are strongly connected with the aforementioned agents architectures. *Agreement technologies* [Oss12], in particular, is a new vision outlining next-generation, open, distributed systems where interaction between computational agents could be based on the notion of *agreement*. This calls for

- a normative context defining the rules of the game, or the “space” of agreements that the agents can possibly reach;
- an interaction mechanism to establish (first) and enact (then) agreements; and
- a joint research effort from several fields – including, but not limited to, multi-agent systems, semantic technologies, social sciences – aimed at fruitfully combining results and contributions from all such areas—like, for instance, semantic alignment, negotiation, argumentation, virtual organisations, learning, real time, and others.

Semantic web standards provide a good basis for representing the knowledge of local agents, the functionalities and everything needed to achieve a goal in agreement with other agents.

However, the formalisms behind these technologies fall short when dealing with the distributed, open and heterogeneous nature of AT systems where agents may have different views of the world and, therefore, mutually-inconsistent knowledge bases. To cope with this issue, new logical formalisms – specifically aimed at handling situations where pieces of knowledge are independently defined in different contexts – have been defined, extending classical logics in order to deal with incomplete and *defeasible* knowledge. Logic is thus exploited to represent the knowledge with a high degree of peculiarity (for instance defeasibility, but also the possibility to discern among permission, obligation or beliefs in deontic logic), and to reason over such knowledge.

Many interesting experiments have been performed in this application area. Notable defeasible argumentation implementations, aimed at supporting reasoning and resolve inconsistencies, are Defeasible Logic Programs [GS04], ASPIC [MP14], and ABA [DKT09]. Several other applications use ontologies and legal search engines [SVJNM16], which exploit advanced search technology from AI, data mining, data analytics, ontologies and natural language processing [KHC18]. The main issues that remain unsolved in this area is that a unique and general framework for dealing with norms and argumentative issues is still missing: in fact, most solutions are too narrow in scope, tailored to specific use cases, other than being possibly weak from the software engineering perspective.

In short, logic-based approaches in the legal field [PS15]

- help formalise legal norms and concepts in a clear and understandable way, thus enabling verification and the detection of unfair policies, or the violation of essential rights;

- support explanatory and arguable decisions in the regulatory context.

In general, however, current tools are unable to imitate advanced cognitive processes such as human reasoning, understanding, meta-cognition or the contextual perception of abstract concepts that are essential for legal thinking. Indeed, a lawyer's work is often very complex, implying the management and processing of huge amounts of data, where to find correlations between facts and circumstances, and formulate reasoned opinions and action guidelines taking into account all the applicable rights and obligations. This is why the process of understanding and formulating a decision is mostly creative—the result of a complex cognitive process.

Education

AI has been part of many e-learning platforms for a long time, with applications ranging from personalised learning, recommendation of resources, automated grading, to prediction of attrition rates—to name just a few. The rapid expansion of the *educational technology* industry is now further pushing and exploiting advanced AI-enabled learning technologies.

Within this area, symbolic AI techniques have been used in adaptive educational systems, such as fuzzy logic, decision tree, etc. there, logic has been mainly applied for knowledge management and recommendation. In some systems, for instance, the focus is on examining and assessing the student characteristics in order to generate students' profiles, to be used for evaluating their overall level of knowledge and, consequently, as a basis for prescribed software pedagogy. Symbolic AI approaches are used to support the diagnostic process, so that the course content can be adjusted to cater each student's needs. Some of them, in addition, are also used to learn from student's behaviour so as to adjust the prescribed software pedagogy.

Applications are related to semantic web technologies, contextualised to e-learning so as to adapt instruction to the learner's cognitive requirements in three ways—background knowledge, knowledge objectives and the most suitable learning style [GFHS04, SMFM05]. Over the years, fuzzy logic techniques and logic MAS have also been experimented for e-learning purposes. In particular, in [AH13, CV13], a fuzzy logic-based system learns the users' preferred knowledge delivery to generate a personalised learning environment; whereas in [GC06] agents detect, recognise, eliminate, and repair the faults of the e-learning course, keeping the system up & working, providing robustness. Another interesting application in the domain of formal logic proofs, taken as the base of several further extensions, is the Logic-ITA (Intelligent Teaching Assistant) web-based system [Yac05]: its purpose is to soothe the issues caused by large classes or distance learning, acting as an intermediary between teacher and students. On the one hand, it provides students with an environment to practice formal proofs, giving proper feedback; on the other, it allows teachers to monitor the class's progress and mistakes.

Although the impact on classrooms has been relatively minor so far, the potential of AI in education is high and likely to increase, as demonstrated by the many European actions / projects currently in place [PSRV19, Tuo18]. The main challenges and issues concern the creation of a sustainable educational environment, capable of developing equitable education even for the least developed countries—to be dealt with at the suitable political level.

7.1.3 AI for Business: Automation & Robotics

Automation is probably the earliest and perhaps most impactful application area for AI, as it represents the first step towards machine autonomy. Autonomy, in turn, is highly desirable whenever there is a need of re-designing, re-building, or re-programming machines while the deployment context evolves. *Autonomous* machines differ from *automatic* ones in that designers no longer need to forecast any possible situation, because the machine is programmed for learning or planning. This is particularly interesting in *cyber-physical systems* (CPS) [BG11] and *robotics*, where machines have a physical body that makes them capable of affecting (or being affected by) the physical world.

Needless to say, applications of automation and robotics in industry are manifold, and so are the corresponding research lines. In the following we explore the role and impact of logic-based paradigms and technologies in such areas.

Planning & Task Allocation

Planning and scheduling are one of the oldest fields in AI, also related to multi-agents and cognitive architectures: research has mainly to do with the decision making process that determines what, when, where, and how to reach a goal and compute a task. There, logic is exploited to represent the knowledge domain, its constraints, and the reasoning mechanism.

Logic-based scheduling methodologies include *rule-based* approaches and *constraint-guided* search. Rule-based scheduling methods aim to emulate the decision-making behaviour of human schedulers, captured in terms of suitable logic rules. Correspondingly, rule-based systems are typically envisioned as a means to replicate the actions of experienced humans with specific scheduling skills. Unsurprisingly, this is one of the most successful application domains of CLP techniques: the scheduler goal is to identify feasible solutions which balance different constraints or schedule requirements.

Applications spread from manufacturing to traffic scheduling and management (e.g., autonomous vehicles, aircraft, ...), up to urban search and rescue activities (e.g., traffic assignment in natural disaster evacuations, . . .), and many others. A typical example of a constraint-based scheduling application is ATLAS [AMPV06], which schedules the production of herbicides at the Monsanto plant in Antwerp. PLANE [Sim01] is another system used at Dassault Aviation to plan the production of the military Mirage 2000 jet and the Falcon business jet: the objective is to minimize changes in the production rate, which has a high set-up cost, while finishing the aircraft just in time for delivery. The COBRA system [Sim01] generates work plans for train drivers of North Western Trains in the UK: each week, about 25000 activities need to be scheduled in nearly 3000 diagrams on a complex route network. The DAYSY Esprit project [SCK00] and the SAS-Pilot program [BKC94] consider the operational re-assignment of airline crews to flights. The STP (Short Term Planning) application at Renault [Wal96] assigns product orders to factories so as to minimise transportation costs. The MOSES application by COSYTEC [Wal96] schedules the production of compound food for different animal species, eliminating contamination risks while satisfying customer's demand at the minimal cost. FORWARD [Sim96] is a decision support system, based on CHIP, used in three oil refineries in Europe to tackle all the scheduling problems occurring in the process of crude oil arrival, processing, finished product blending and final delivery. Finally, Xerox has adopted a constraint-based system for scheduling various tasks in reprographic machines (like photocopiers, printers, fax machines, etc.); the scheduler is supposed to determine the sequence of print making and coordinate the time-sensitive activities of the several hardware modules that make up the machine configuration [Bis01].

Overall, the CLP approach faces well some of the key issues such as development time, nodes visited in the search tree, number of generated feasible solutions, and efficiency. At the same time, the very nature of such systems mandates for considerable development and tuning effort for each new application, as there is no expert to emulate. More generally, the main two issues are *i)* the lack of a structured way to carry the insight gained from one application to the next, and *ii)* the complexity of generating the symbolic knowledge that fully describes the application domain.

Robotics & Control

Cognitive architectures, planning, and task allocation techniques have been widely applied to robotics and control system: indeed, robotics applications translate the agent abstraction of cognitive architecture into a mechanical robot capable of doing action and taking decision.

Logic in robotics is, much more than elsewhere, tailored to the specificity of the application field, since control mechanisms need to control the robot sensors and actuators, along with all their low-level control software (for instance, robot motion mandatorily requires a set of feedback control primitives in order to keep motion coherent). More generally, control systems are present in lifts, photocopiers, car engines, assembly lines, power stations, etc.

Again, the logic-based approaches (and technologies) that are mostly adopted in this context are CLP, fuzzy logic, and temporal logic: in fact, many works dealing with robotic reasoning [FL08] exploit languages and technologies detailed in section 5.1. CLP-based applications are typically at the smaller end, where it is still possible to prove that some global properties can be guaranteed with a given control. Thus, CLP is often exploited to build control software for electro-mechanical systems with a finite number of inputs, outputs, and internal states: each component is connected to a small part of the overall system, so its behaviour can be captured quite simply. However, when the system is considered as a whole, the number of global states can become very large: this calls for a smart technology that is able to handle such combinatorial explosion [Lyt08].

On the other side, many control system have been developed exploiting a fuzzy logic approach for dealing with real data which are sometimes imprecise, uncertain, complex, and with a high degree of randomness. Due to its good tolerance of uncertainty and imprecision, fuzzy logic has gained wide application in the area of advanced control of humanoid robots. For the same reason, fuzzy system are powerful tools to face crucial problems in industrial engineering and technology, such as risk management or product quality assurance, as well as in intelligent decision support system for adaptive industrial engineering [CM20, RAS15, PKD$^+$12, ZY04, CV98]. Also, the hybrid techniques based on the integration of neuro-fuzzy networks, neuro-genetic algorithms, and fuzzy-genetic algorithms are of great importance in the development of efficient algorithms [TH12].

Temporal logic approaches and technologies have been exploited, e.g., for controlling robot motion or planning activities [FJKG10, FGKGP09], because of their ability to reason over the time and its change, which makes it possible to build control laws to be verified over the time elapse. This is specially relevant for mobile robots, whose specifications are often temporal—even though time is not necessarily captured explicitly. For example, a swarm might be required to reach a certain position and shape eventually, or maintain a size smaller than a specified value until a final desired value is achieved. Other examples are collision avoidance among robots, obstacle avoidance, and cohesion, which are always required. In a surveillance mission, a selected area needs be visited “infinitely often” [KB07].

7.2 Discussion

The new AI era calls for two fundamentals enabling factors: *(i)* the availability of big computing power even in minimal spaces, and *(ii)* the availability of huge amounts of context-related data. Such factors on the one side make it possible to learn from experience, which is the playground of sub-symbolic algorithms; on the other, make logic algorithms, historically used for expert systems and hence very effective as for transparency and explainability, computable *in reasonable time*.

Big data have naturally led sub-symbolic approaches to prevail, because of their effectiveness in elaborating and getting valuable results from context-related data, learning trends and repeating patterns: yet, their inherent black-box nature is a clear issue. This is precisely where the integration with symbolic approaches can naturally provide added-value, complementing symbolic and sub-symbolic approaches with each other. In fact, the two main weaknesses of symbolic approaches concern specifically *(i)* the extraction of context-related knowledge, and *(ii)* computational complexity—the first being the natural territory of sub-symbolic approaches.

Computational complexity, in its turn, can be partially addressed thanks to the exponential increase of computational power, by suitably exploiting parallelism, as well as by re-defining and re-tuning symbolic approaches so as to fit nowadays computing paradigms and architectures. However, it still remains an issue in some application contexts, often leading to higher processing costs. An example is propositional interval temporal logics (ITL) [GMS04], which provide a natural framework for representing and reasoning about temporal properties, but whose computational complexity constitutes a barrier for extensive use in practical applications. To cope with this issue, several approaches exploit constraints or adopt a locality principle; in other cases, as in DLs, complexity is decreased at the price of a limited expressiveness. For instance, in Bowman and Thompson's decision procedure for propositional ITL, decidability is achieved by means of a simplifying hypothesis – the locality principle – that constrains the relation between the truth value of a formula over an interval and its truth values over initial sub-intervals [HLW08]. Tableau-based decision procedures have been recently developed [DMGM$^+$13] for some interval temporal logics over specific classes of temporal structures, without resorting to any simplifying assumption.

Parallelism and concurrent programming techniques are another valuable tool to deal with complexity. The computing power of multi-level parallelism (MLP), in particular, constitutes a promising technique to facilitate concurrent programming while delivering performance comparable to that of fine-grained locking implementations—see for instance [LR07] and [JS13].

To recap, due to their strong foundations and features, logic-based technologies have the full potential to power symbolic approaches in such integration, opening intriguing perspectives currently under exploration in many research contexts.

Overall, the synergy of symbolic and sub-symbolic approaches appears to be a viable and promising option to face key issues in today's intelligent systems—

namely, the need of explainable, responsible, ethical AI. In particular, the adoption of symbolic approaches can help to achieve the key features of e-justice, fairness, ethics and transparency.

The role of technology. It is worth to be highlighted how in this chapter we mostly focus on the *role* of logic based technologies from an utilitarian perspective, i.e. in terms of what functionalities they may offer and what applicative scenarios they may serve. However, in practice, a key role in the rise (or fall) of a technology lays in a number of technical aspects. These include *(i)* its ease of use, *(ii)* the problem it solves, as well as *(iii)* the problems it indirectly helps solving, *(iv)* how it interoperates with other well established technologies, and therefore *(v)* the value it adds to (and receives from) them, other than *(vi)* its resilience w.r.t. the fast-paced change which is inherent in software technologies—i.e. whether it is actively *maintained* or not.

To further asses all such aspects, in the next chapter we analyse the state of the art of logic-based technologies from a technical perspective. In doing so, we analyse the maturity, technical reach, and maintenance-level of most relevant logic-based technologies.

Chapter 8

Technological State of the Art

This chapter contains contributions from the following works of ours: [CCMO21a, CCMO21b]

Precisely when the success of artificial intelligence (AI) sub-symbolic techniques makes them be identified with the whole AI by many non-computer-scientists and non-technical media, symbolic approaches are getting more and more attention as those that could make AI amenable to human understanding. Given the current status of AI technologies – mostly focussed on sub-symbolic approaches successful in well-delimited application scenarios –, a key issue for intelligent system engineering is *integration* of the diverse AI techniques: in software engineering terms, not just how to integrate diverse technologies, but also how to preserve *conceptual integrity* when highly-heterogeneous approaches – bringing about manifold abstractions of various nature – are put to work together.

The most straightforward and generally-acknowledged way to address the above issue is by using *agents* and *multi-agent systems* (MAS). Agents and MAS have been at the core of the design of intelligent systems since their very beginning: their long-term connection with *logic-based technologies* might open new ways to engineer *explainable intelligent systems*.

This is why understanding the current status of *logic-based technologies for MAS* is nowadays of paramount importance and why our work focus on logic-based approaches in MAS: they are to be counted among the most promising techniques for building understandable and explainable intelligent systems. Furthermore, given the unavoidable push towards the exploitation of intelligent applications, focussing on logic-based *technologies* is of strategical importance. Accordingly, understanding and representing the current status of the available logic-based MAS technologies is a key step – from both an historical and an avant-garde perspective – to let MAS researchers and practitioners identify the actually usable methods for the engineering of intelligent systems.

To this end, in [CCMO21a] we provide a Systematic Literature Review (SLR) driven by the primary research question: "What is the role played by logic-based technologies in MAS nowadays?". In particular, the SLR aims at understanding which and how many logic-based technologies for MAS can be considered ready enough to face the challenges of modern and future intelligent systems, other than identifying what is missing and what research directions require further attention. Accordingly, the goal is to provide an exhaustive assessment of the available logic-based technologies for MAS, by performing a carefully-designed SLR on the subject.

8.1 Method

The SLR follows a well-founded, understandable, and reproducible method defining how to find, include/exclude, and analyse papers describing logic-based MAS technologies. It relies on the standard SLR method: we carried out a manual retrieval, filtering, analysis, and categorisation of huge number of papers, by repeating 8 queries on 6 search engines (Google Scholar, IEEE Xplore, ScienceDirect, SpringerLink, DBLP, ACM Digital Library) and 5 specific conference/workshop proceedings. To keep a tight focus on the *reproducibility* of the whole process, the methodological approach, and the inclusion/exclusion and analysis criteria are carefully designed and described in detail. In particular, we only included works defining or exploiting some *logic-based MAS technology.* A specific definition of *logic-based MAS technology* is provided as well, explicitly requiring the provable availability of *(i)* a clearly identifiable logic-based MAS-related framework into the literature, and *(ii)* some actual software reification of that framework.

Out of the retrieves papers, we selected 271 documents and there identified 47 technologies, classified them according to both a MAS and a logic perspective, and analysed from the technology viewpoint. Accordingly, the technologies selected in our SLR are analysed and assessed from two different perspectives – namely, the MAS and the logical perspective –, thus discussing the specific MAS- and logic-related aspects defined, tackled, or exploited by each technology. Along the MAS perspective, we categorise the selected technologies w.r.t. the main MAS abstractions they relate to—agents, societies, environment. Along the logic perspective, we categorise the selected technologies w.r.t. the sort of logic they relate to—thus choosing among first-order logic, description logic, BDI logic, etc. Such categorisations reveal an uneven distribution of logic-based technologies along both MAS abstractions and logics, and highlighting research opportunities on abstractions and logics which are currently in an urgent need of technologies—such as the environment abstraction and the defeasible logic.

We also perform a technical assessment of each technology, according to number of technological criteria including, but not limited to, *(i)* source-code organisation, *(ii)* maintenance status, *(iii)* target platform(s), *(iv)* availability of executable as well as documentation, *(v)* some technical assessment involving the run of executables and available examples. Arguably, the analysis enables a detailed discussion on the current state of logic-based MAS technologies—in particular highlighting their state of maintenance. More precisely, we consider technologies as unmaintained based on the last provable edit involving either the technology source code or any of its software artefacts[1].

8.2 Detailed Technological Analysis

In this subsection we deepen the analysis of the technologies selected in [CCMO21a], in order to provide additional insights about research questions. Table 8.1 provides an overview of our analysis.

There, we analyse the selected technologies according to a number of technical dimensions, corresponding to the columns of table 8.1. In particular: column **Fresh.** assesses the *freshness* of each technology, by indicating – when possible – the year of the visible update; column **Code** provides a qualitative assessment of each technology *code base*, when available; column **Doc.** provides a qualitative assessment of each technology *documentation*, when available; column **License** indicates under which license each technology is provided, if any; column **Target** reports on the target *runtime platform(s)* each technology can be executed upon; column **Runs** shows our success in *executing* each technology – or loading it into its target runtime –, possibly after any necessary compilation step; column **Benchmark** points out whether each technology is distributed with some *benchmarks* or *examples*; column **Work** points out whether each technology *works*, i.e., if the aforementioned benchmarks / examples can be executed successfully.

More precisely, the freshness of a technology is defined by the *year* of its most recent release, or, source code modification. The code-base assessment consists of a (possibly missing) integer number ranging from 1 to 5:

- "1" means the code base is available as a bare archive, even if it appears to have no clear file organisation, and it is comes with no facility supporting the compilation (or, in general, usage) of the code;

- "2" means the code base is available as a bare archive, and it adheres to a well organised structure or it includes instructions on how to compile/use it

[1]Since our original assessment (September 2020), some technologies described as unmaintained reworked their repositories, code and/or documentation—as in the case of DALI (`https://github.com/AAAI-DISIM-UnivAQ/DALI`) and MCAPL (`https://autonomy-and-verification.github.io/tools/mcapl`)

Table 8.1: Overview on the selected technologies and their analysis. Dashes represent missing values, whereas question marks represent unknown values.

Name	**Fresh.**	**Code**	**Doc.**	**License**	**Target**	**Runs**	**Benchmark**	**Works**
2APL	2012	-	3	none	JVM 6,7,8	Yes	*.mas	Yes
2CL	2013	-	1	none	JVM 7	Yes	*.2cl	Yes
3APL	2007	-	4	none	JVM 6	No	-	-
Agent-0	1993	1	1	none	(Allegro) Common Lisp	Yes	none	No
AFAPL2	2016	3	-	LGPL v2	JVM	?	none	?
AgentOWL[a]	2013	3	1	LGPL v2	JVM 8, 11+	Yes	Compiled demo	No
AIL	2018	3	1	LGPL v3	JVM 8+	Yes	*.ail	Yes
ALIAS[b]	2000	1	1	none	JVM | Prolog	No	-	-
AORTA	2015	4	-	none	JVM +7	Yes	*.ail	Yes
ASPARTIX[c]	2018	-	1	MIT	Web Service	Yes	*.dl	Yes
ASPIC	2019	4	3	LGPL v3	JVM 12+	Yes	Compiled demo	Yes
Astra[e]	2018	3	5	GPL v3	JVM	Yes	Provided examples	Yes
ConGolog	-	-	-	?	?	?	?	?
DALI	2018	3	1	Apache v2	SICStus	Yes	?	?
DCaseLP / DyLog	2005	2	3	none	JVM	No	-	-
DeLP[d]	2018	4	3	LGPL v3	Web Service	No	-	No
DRAGO	2006	2	2	none	JVM 5+	Yes	Any OWL ontology	Yes
EMERALD[a]	2010	-	3	none	JVM 6+	Yes	Compiled demo	Yes
eXAT	2012	4	3	GPL v3	Erlang	No	-	No
Go!	2015	2	-	GPL v2	?	?	?	?
GOAL[e]	2019	5	4	GPL v3	JVM 8	Yes	Template projects	Yes
Golog	1998	1	-	ah hoc	SWI | ECLiPSe	Yes	-	?
IndiGolog	-	-	-	?	?	?	?	?
JACK	-	-	5	proprietary	JVM	?	?	?
Jadex	2020	5	5	GPL v3	JVM 8+	Yes	Compiled demo	No
JADL/JIAC	2018	4	5	Apache v2	JVM 7+	Yes	Provided examples	Yes
Jason	2020	5	5	LGPL v3	JVM 8+	Yes	Provided examples	Yes
Jason-ER	2019	5	-	LGPL v3	JVM 8+	Yes	Provided examples	Yes
jDALMAS	2016	3	-	none	JVM	No	-	-
LPaaS	2018	5	3	Apache v2	JVM 8+	Yes	none	?
MCAPL	2018	3	1	LGPL v3	JVM 8+	No	*.ail	No
MCK	-	-	3	none	Web Service	Yes	Provided examples	Yes
MCMAS	2017	4	3	none	Native / Linux	Yes	From manual	Yes
Mozart	2019	5	5	ad-hoc	Native	Yes	Provided examples	Yes
MetateM	2010	2	3	GPL v3	JVM 6+	Yes	*.sys	Yes
Onto2Jacamo[e]	2017	-	1	none	JVM	No	-	-
Raspberry	-	-	1	none	Native / Linux	Yes	*.rasp	Yes
RML	2019	5	3	Apache 2	JVM 8+	Yes	*.rml	Yes
Rodin	2020	4	?	none	JVM 8	Yes	none	?
SALMA[f]	2016	3	1	none	Python	No	-	-
SCIFF	2008	1	2	none	SWI | SICStus	Yes	none	?
SHOP	2020	4	2	MPL	Common Lisp	Yes	?	?
Spindle	2017	2	4	GPL v3	JVM 7+	Yes	*.dfl	Yes
Teleo-R/QLog	2019	3	3	none	QuProlog	No	-	-
TuCSoN / Respect (and variants)	2020	3	3	LGPL v3	JVM 8+	Yes	Compiled demo	Yes
TuSoW	2020	5	-	Apache 2	JVM 8+	Yes	none	?

[a] depends on some ancient Jade version, which is not provided. [b] requires the Jinni Prolog interpreter for Java. [c] requires the Clingo solver. [d] depends on ancient tuProlog version, which is not provided. [e] is a plug-in for (or customisation of) the Eclipse IDE. [f] requires ECLiPSe Prolog and the PyCLP Python module.

- "3" means the code base is available through some version control system [2] (VSC, henceforth), but poor support is provided for compilation or usage;
- "4" means the code base is available through some VCS and it comes with some build automation tool[3] as well, supporting compilation or usage;
- "5" means the code base is available through some VCS, it comes with some build automation tool, and it is also distributed through some official repository;

whereas a missing value denotes that no assessment can be drawn given the available information—e.g., because we were not able to access any code base.

Similarly, the documentation assessment consists of a (possibly missing) integer number ranging from 1 to 5 as well. In this case, however:

- "1" means that the only available form of documentation is some textual note briefly describing the technology or how manage the codebase;
- "2" means that some *structured* form of documentation exists describing the technology or how manage the codebase;
- "3" means that a *detailed manual* or some *API reference* are available, but *not both*;
- "4" means that *both* a detailed manual and some API reference are available;
- "5" means that a detailed manual, some API reference, and some *usage examples* or *tutorials* are available;

whereas a missing value denotes the total lack of any form of documentation.

Whenever available, the license of each technology is referenced in table 8.1 as well, possibly leveraging on well-known license acronyms. For instance, as far as open source licenses are concerned, "GPL" refers to the GNU General Public License[4], "LGPL" refers to the GNU *Lesser* General Public License[5], "MIT" refers to the MIT License[6], "MPL" refers to the Mozilla Public License[7], whereas "Apache" refers to the Apache License[8]. The absence of licenses for a particular technology is pointed out as well, as it may have an impact on its users[9].

[2]e.g., SVN, Git, Mercurial, etc.

[3]e.g., Make, Apache Ant, Apache Maven, Gradle, Pip, Npm, etc.

[4]`https://www.gnu.org/licenses/gpl-3.0.html`, last accessed in April 2020.

[5]`https://www.gnu.org/licenses/lgpl-3.0.html`, last accessed in April 2020.

[6]`https://opensource.org/licenses/mit-license.php`, last accessed in April 2020.

[7]`https://www.mozilla.org/en-US/MPL/2.0`, last accessed in April 2020.

[8]`https://www.apache.org/licenses/LICENSE-2.0`, last accessed in April 2020.

[9]`https://choosealicense.com/no-permission`, last accessed in April 2020.

The target runtime platform is another relevant aspect we analyse for each technology. It provides an intuition of which sorts of machines and devices could in principle be capable of running a given technology. As it clearly emerges from table 8.1, the JVM platform is targeted by most technologies. This is why, in the particular case of JVM-based technologies, we try to assess the specific version(s) of the JVM they can run upon. Thus, the "JVM N+" notation indicates that a given technology is tested on all JVM versions ranging from N (included) to version 13 (included) – which is the most recent one at the time of writing –, and only executes without errors starting from version N. Of course, the JVM is not the only platform our selected technologies leverage upon. Some technologies require a compilation step targeting some *native* platform. So, for instance, in case only the *OS* operative system is supported, we write "Native / *OS*" to identify the target platform. Other technologies target the Common Lisp, Python, or Erlang runtimes, which come with several implementations supporting mainstream operative systems, similarly to what JVM does. Furthermore, a few technologies are available as web services. In those cases, we argue that the actual platform of the service implementation is not essential—this is why we simply denote the target platform as "Web Service". Finally, there are some technologies which are explicitly aimed at extending (or customising) the Eclipse IDE [10], and are not meant to be used otherwise. In those cases, we indicate "JVM" as the target platform, and tag the technology through an *ad-hoc* footnote.

In order to test whether a technology runs or not, we simply launch it on its target platform—possibly, after performing all necessary compilation/configuration steps. If neither compilation/configuration nor launching produces error or crash, then we say the technology runs, otherwise it does not. Thus, a question mark in table 8.1 in the **Runs** column may indicate either the total lack of any information on how to launch the technology, or, the impossibility of producing / accessing an executable to launch.

Finally, when we are able to run a given technology, we then test it to get further detail, so as to understand if it actually *works* or not. To do so, we first look for available *benchmarks* or examples into the running technologies code bases, documentation, or home pages. In case some benchmarks / example are available, we check if they can be run without producing error outputs or crashes. If they can, then the technology works, otherwise it does not. Thus, a missing value in table 8.1 in the **Benchmark** or **Works** columns indicates that no assessment is needed because the technology does not run. Conversely, a question mark in the same columns denotes the impossibility to perform any further assessment due to lacking benchmarks, examples, or instructions on how to launch them.

By aggregating the data in table 8.1, we can draw several interesting con-

[10]`https://www.eclipse.org/ide`, last accessed in April 2020

Table 8.2: Statistics on selected technologies

	Absolute	**Relative**	**Meaning**
N. selected tech	47	100%	
Maintained since 2019	12	25.53%	Fresh. $\geq$ 2019
Maintained since 2018	20	42.55%	Fresh. $\geq$ 2018
Open source	26	55.32%	License $\notin \{\text{none}, ?\}$
Unlicensed	19	40.43%	License = none
JVM-based	30	63.83%	Target starts with JVM
Certainly runs	31	65.96%	Runs = yes
Certainly works	21	44.68%[a]	Works = yes
Codebase quality	16	34.04%	Code $\geq$ 4
Documentation quality	9	19.15%	Doc. $\geq$ 4

[a] corresponding to 67.74% of the technologies which certainly run.

clusions. Most relevant ones are summarised in table 8.2. The most evident information is that only 12 out of 47 technologies have been actively maintained since 2019—i.e., 25.53% of the total. The percentage is lower than 50%, even if we enlarge the spectrum to technologies maintained since 2017. However, most technologies (65.96%) can still be run successfully – regardless of when they were last updated –, even though we are able to make them work in 67.74% of cases only.

Another interesting trait is that – except for Jack – all technologies that are explicitly licensed come with an open source license. These correspond to 55.32% of the total. The amount of unlicensed technologies is quite high as well, as it corresponds to the 40.43% of the total.

It is interesting to note how the JVM is by far the preferred platform for logic-based MAS technologies. Indeed, 63.83% of the selected technologies target some version of the JVM. Other recurring platforms are Lisp-, Prolog-, or native-based.

Finally, it is worth to be mentioned how – except for a few notable exceptions, such as Jason – poor care is given to technologies code bases and documentary resources. Indeed, considering the 1-5 ranges defined above, only 34.04% of the technologies come with a code base whose quality is greater than 3, whereas only 19.15% are scored similarly as far as documentation is concerned.

8.3 Main Outcomes

The outcome of the SLR highlights that, as far as logic-based technologies for MAS are concerned, there is still room for technological advancements—except for a few relevant success stories. In fact, despite the enormous technological effort clearly carried out by the MAS community in the last decades, several surveyed technologies cannot be considered as mature and ready for use in the new challenging contexts required by AI. Several technologies are in fact unmaintained, outdated, or just proof of concepts.

In our original work the discussion attempts to provide a comprehensive answer to all the SLR research questions. In the following we summarise some general remarks in relation to key features of modern intelligent systems, namely: *(i)* inherent distribution and decentralisation and deep entanglement with domains like the Internet of (Intelligent) Things (Io(I)T) and Cyber-Physical Systems (CPS); *(ii)* support to key properties such as robustness, efficiency, interoperability, portability, standardisation, situatedness, and real-time support; *(iii)* need to reconcile and synthesise symbolic and sub-symbolic AI, exploiting the former to *explain* the latter so as to overcome fears and ethical issues posed by AI by providing for explainability, observability, interpretability, responsibility, and trustability—the scope of XAI.

Applicability to distributed domains such as IoT and CPS

The existing agent-oriented logic-based solutions applicable to the IoT and CPS are only available for a specific and limited set of devices and platforms. For instance, Agent Factory Micro Edition (AFME) [MOCO06] enables the execution of a deliberative agent on top of mobile phones with CLDC/MIDP profiles and Sun-SPOT sensor by means of TCP/IP and Zigbee protocols.

However, some technologies, more than others, are explicitly designed to support IoT domains and CPS. For example, the LPaaS architecture [CDMO18] is designed to promote distributed intelligence for the IoT world—offering logic programming as a service, and explicitly addressing the requirements and issues of cloud and edge architectures. Analogously, the situated coordination approach promoted by the TuCSoN/ReSpecT model and technology can be explicitly exploited to handle situatedness in MAS as a coordination issue. Also, TuCSoN [OZ99] provides the main abstractions for IoT environments: environmental resources can be sources of perceptions (like sensors), targets of actions (like actuators), or even both.

Finally, there are technologies that are not explicitly meant to address the IoT and CPS domains, but still let us suppose they would be easily portable to those domains—because of their standard compliance, interoperability, and portability features. Among the many, Jason [BH06] supports interoperability with non-Jason agents via JADE [BBCP] through FIPA-ACL communication [FIP02]. Similarly, there are extensions to JACK [Win] that make it work in open systems. Finally, the Teleo-Reactive [Nil01] approach has been often exploited to facilitate the development of the IoT systems as a set of communicating Teleo-Reactive nodes.

Symbolic and sub-symbolic integration

With respect to the need to reconcile and integrate symbolic and sub-symbolic techniques, none of the selected technologies has been experimented yet [CCO20], due to their original design purpose out of this scope. However, we argue that portable and interoperable technologies might be more suitable for the integration. Anyway, the field is still unexplored and represents a frontier research domain.

Can existing technologies be labelled as ready? If not, what is missing?

The role of logic-based technologies in MAS nowadays exhibits a huge potential for covering the vast majority of intelligent system abstractions. However, just a few among the technologies surveyed can be actually labelled as *ready-to-go*, in particular when considering the new challenges for symbolic technologies in AI.

Even though 10% of the selected technologies can be considered as mature – in terms of cross-platform support, code quality, and ease of distribution in heterogeneous environments –, most of the times they have not been tested in pervasive and real-world scenarios, yet. This implies, at least, that further research and technical activity are required to ensure that any technological barrier can be overcome. Furthermore, integration with sub-symbolic techniques remains a *nice-to-have* feature, but it is not actually a thing in any MAS technology, for the time being. Nevertheless, the selected technologies are an excellent starting point for *(i)* highlighting the advantages of logic-based technologies, and *(ii)* broadening the scope of research towards the directions envisioned.

The Need for Interoperability

Our survey identifies two major lack in the logic-based technology playground, namely *maintenance* and *interoperability*.

Maintenance (and its lack) is essentially an organizational aspect concerning individual research groups around the globe. Career incentives within academia discourage the maintenance of software on the long run. So, the current maintenance status of logic-based technologies is unsurprising. In this perspective, we consider as positive that a non-negligible percentage of logic-based technologies has a decades-long life span.

The actual issue we detect through our survey concerns *interoperability* among the logic-based technologies which survived so far. Arguably, this is the most relevant aspect research communities should be concerned by. Existing logic-based technologies often address specific uses cases, or rely upon monolithic runtimes, often targeting single platforms or having inconvenient constraints and dependencies. Poor care is devoted to favour their interoperability and joint exploitation, probably because of the different programming platforms/runtimes they leverage upon, or the use cases they have been designed for. In other words, logic-based technologies are often constructed as technological *silos* – being so optimised for performance and correctness while being poorly interoperable among each other – targetting the LP or MAS communities alone.

On the other side, the data science playground is flourishing, also because of the emergent selection few key platforms – e.g. Python or R – upon which many inter-dependent and interoperable research project are built, adding value to each other.

Hence, the need for efforts pushing the state of the art of logic-based technologies towards a higher degree of interoperability – both internally, and with DS technologies – is compelling. Along this line, in the next chapter, we propose a paradigm shift aimed at satisfying this need. In particular, we propose the design and implementation of a general-purpose *logic-ecosystem*, upon which other logic-based (as well as ML-oriented) technologies can be constructed, embedded, or made interoperable with. Then, in the remainder of this thesis, we discuss how our ecosystem can be designed and extended to cover the need for interoperable technologies at the intersection among symbolic and sub-symbolic AI.

Chapter 9

The 2P-Kt Ecosystem for Logic-Based AI

This chapter contains contributions from the following works of ours: [CCO21a]

This chapter contains contributions from the following Master's thesis: [Sib19], which we supervised

While their impact on the *symbolic* branch of AI is well established, many emergent AI techniques leverage logic to make data-driven AI either more predictable or more understandable. This is why the need for solid, interoperable, general-purpose *logic-based technologies* is nowadays more compelling than ever. Most of the logic-based technologies proposed so far are typically either built on top or as extensions of the Prolog language. Even when this is not the case, *monolithic* solutions are built around different inference procedures, unification mechanisms, or knowledge representation techniques.

This chapter stems from the idea that logic-based technologies should be neither constructed as Prolog-centred monoliths nor tailored to a specific semantics or language. Instead, in order to maximise their impact on AI, logic-based technologies should welcome the manifold contributions coming from the LP playground, supporting the exploitation of as many mechanisms as possible, in an unopinionated way. As a foundational step in that direction, we present 2P-Kt, a reboot of the tuProlog project offering a general-purpose, extensible, and interoperable *ecosystem* for logic programming and symbolic AI.

9.1 The Need for an Ecosystem

In spite of the wide availability of logics, inference rules, and resolution strategies in the LP literature, only a relatively small amount of them have been reified into actual logic-based technologies. Among these, the Prolog language [CR93] is by far the most successful story [CCDO20]. It consists of a well-defined language

[IS95, ISO00] coming with several implementations [BPr21, Pro21a, Pro21b, SP21, Pro21c, Pro21e, Pro21d].

While standard implementations of Prolog target first-order logic (FOL) via SLDNF inference rule [Rob65, Kow74, Cla77] and depth-first resolution strategy, most implementers have extended Prolog to support other resolution strategies as well. This is the case of Prolog implementations supporting for instance, *constraint logic programming* (CLP) [JL87], *constraint handling rules* (CHR) [Frü98], *tabled resolution* [SW12], etc.

Thanks to the versatility of FOL, it is a common practice in LP to either develop logic-based technologies either *on top* of Prolog or *from scratch*. In the former case, the resulting logic-based technologies tend to be poorly interoperable – as strictly Prolog-dependent [CCS+20] –, while in the latter case they tend to be very narrow in scope—as heavily tailored on a particular domain.

Building logic-based technologies on top of Prolog is often preferred as they automatically inherit Prolog basic mechanisms, including e.g. the capability of *(i)* representing data structures via logic terms, *(ii)* knowledge via Horn clauses, *(iii)* logic unification, *(iv)* efficient in-memory indexing of logic information, *(v)* a flexible inference rule, and *(vi)* meta-level programming. This is the smartest strategy when novel logic-based technologies must be quickly bootstrapped, yet it may easily result in poorly-interoperable, Prolog-tailored solutions. Conversely, when Prolog capabilities are not adequate for the particular problem at hand, logic-based technologies may be designed from scratch. This commonly involves re-designing and re-implementing most LP features from scratch.

In [SY96], Sterling states that logic unification is by itself the major contribution of LP to software engineering—thus singling a specific feature out of Prolog for its value and benefits. Along this path, we argue that many aspects of LP may be useful in AI by themselves, and each contribution could be conveniently reified into some individually-usable software library. Accordingly, our work aims at the creation of an *open ecosystem* for interoperable, general-purpose LP libraries, virtually supporting multiple logics, inference rules, and resolution strategies, and possibly factorising any shared aspect—e.g. terms and clauses representation, unification, in-memory storage, (de)serialisation, etc.

In the early 2000s, the idea of LP as a key *technology*-enabler of intelligent application was already in place. The tuProlog project [DOR01] was proposed for this purpose. It consists of a lightweight *malleable*, object-oriented, Java-based implementation of Prolog [PBOR08] which can be used as a library for JVM projects. Despite many versions have been proposed along the years – bringing new features, or more platforms support [DOC13] –, and many research products have been built upon it – such as TuCSoN [OZ99], ReSpecT [OD01], LPaaS [CCM+18], or Tenderfone [CMOZ20], Arg2P [PCOS20], etc. –, it still consists of a *monolithic* library targetting Prolog *alone*.

Accordingly, this chapter stems from the idea that Prolog is not the silver-bullet for logic-based technologies, and the belief that LP should not be reduced to Prolog alone. For this reason, we present 2P-Kt, a *reboot* of the tuProlog project aimed at providing a common technological ground for LP. Acknowledging that most mechanisms in LP have the potential to be of general value, not necessarily tailored to any specific logic, inference rule, or resolution strategy, 2P-Kt consists of a logic-based *ecosystem* for symbolic AI, designed and implemented by taking openness, modularity, extensibility, and interoperability into account.

More precisely, the tuProlog project has been completely re-designed and re-written, splitting LP functionalities into minimal, loosely-coupled, Prolog-agnostic, individually-usable, multi-platform *modules*. The rationale behind this choice is to enable the incremental addition of novel LP functionalities to the 2P-Kt ecosystem – possibly targeting other inference rules and search strategies –, minimising duplication of features and reusing pre-existing ones, while supporting as many programming platforms as possible.

On the long run, 2P-Kt aims at becoming a comprehensive technological playground supporting several sorts of logics – e.g. first-order, higher-order, temporal, deontic, etc. – and mechanisms—e.g. deductive, inductive, abductive, probabilistic reasoning, etc.

Finally, a non-negligible effort is devoted to keep 2P-Kt widely interoperable at the technological level with as many platforms as possible—to maximise the pool of potential adopters. This is why most 2P-Kt modules are Kotlin Multiplatform projects – currently supporting the JVM, JS, and Android platforms – while others are expected to be supported soon—e.g. MacOS, iOS, .NET, and Python.

9.2 Overall Design

2P-Kt is deeply rooted in CL, a programming paradigm based on computational logic [Llo90, MN96]. Hence, from an architectural perspective, 2P-Kt is a framework supporting the creation of logic-based software via several loosely-coupled *modules*—each one tailored on a particular aspect of CL.

To further support reusability, each module factorises a small number of related functionalities, via a compact API composed by OOP types and methods. As modules are the most basic deployable units in 2P-Kt, major LP functionalities are partitioned into modules on a per-usage basis, to make them selectively usable as dependencies in other projects. The 2P-Kt ecosystem itself is attained by incrementally combining such modules, as depicted in fig. 9.1. Accordingly, to maximise interoperability, 2P-Kt modules are individually available

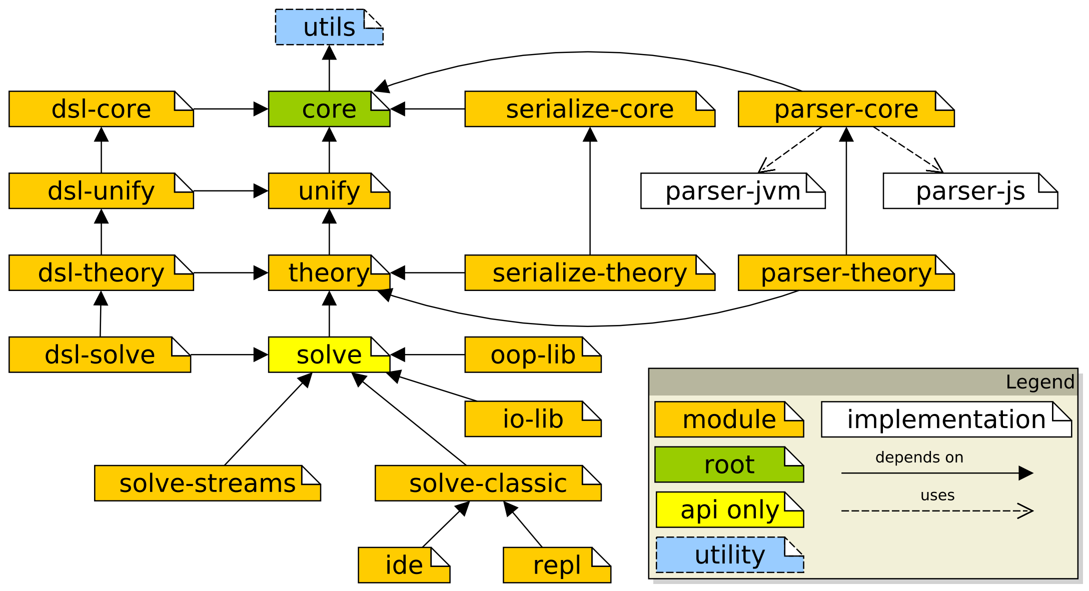


Figure 9.1: 2P-KT project map: LP functionalities are partitioned into some loosely-coupled and incrementally-dependent modules

as pre-compiled libraries both on Maven Central Repository (MCR)[1] – for JVM-, Android- or Kotlin-based contexts – and on the NPM Registry[2] – for JS-based contexts –, whereas a detailed description of their API is available on the Web as part of 2P-KT documentation.

If all 2P-KT modules were merged together, the most relevant aspects of their API could be summarised as in fig. 9.2. The diagram points out how all relevant aspect of LP are reified into types: e.g.

- logic `Terms` (as well as any specific sort of term, e.g. `Variables`, or `Structures`, etc.),
- logic `Substitutions`, unification, and MGU (computed by an `Unificator`),
- `Clauses` (there including `Rules`, `Facts`, and `Directives`),
- knowledge bases and logic theories (e.g. the `Theory` type),
- automatic reasoning, via the `Solver` interface, and
- logic `Solutions`—computed by `Solvers` in response to queries.

[1]`https://search.maven.org/search?q=g:it.unibo.tuprolog`
[2]`https://www.npmjs.com/org/tuprolog`

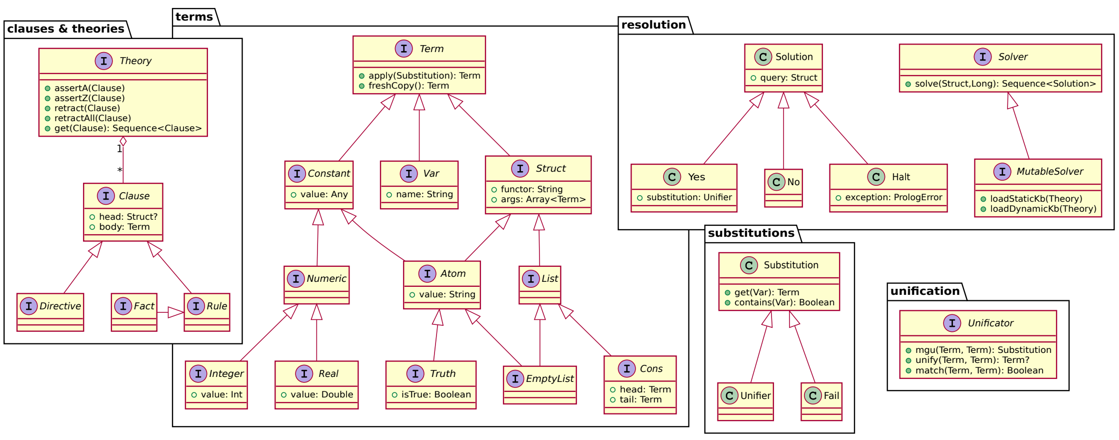


Figure 9.2: 2P-Kt public API: a type is provided for each relevant concept in LP

There, interfaces are used to expose relevant aspects, in order to keep the system extensible. Developers may e.g. define custom implementations for `Unificator` and `Solver` to provide novel inference mechanisms involving some variant of the unification algorithm. Of course, a detailed diagram would include several more types, as the 2P-Kt API also supports: *(i)* (de)serialisation of logic terms and theories into/from standard data-representation formats such as JSON, or YAML; *(ii)* parsing/formatting logic terms and theories from/into concrete logic syntaxes such as Prolog's one; *(iii)* letting developers extend solvers via libraries of custom LP functionalities; *(iv)* letting users exploit logic solvers either by a command-line (CLI) and graphical (GUI) user interface; etc.

9.2.1 Overview of Functionalities

Each major 2P-Kt functionality is reified into a particular module. Accordingly, in this subsection we enumerate 2P-Kt functionalities on a per-module basis. Following Gradle convention, we denote modules by *`:moduleName`*.

The most fundamental module is *`:core`*, which exposes types for representing symbolic knowledge via terms and clauses, other than methods to support their manipulation (e.g. construction, unfolding, scoping, formatting, etc.) for OOP or FP programmers. It comes with several data structures aimed at covering most common KR needs in LP. However, novel sorts of terms and clauses may optionally be added by developers by extending/implementing any public interface in *`:core`*. Furthermore, the pervasive adoption of an *immutable* design makes data structures in *`:core`* well suited for concurrent and multi-threaded contexts.

Terms and clauses are often compared or manipulated in LP via *unification*. Arguably [SY96], unification [BS01] is – by itself –, among the most useful contributions of LP to AI. For this reason, we encapsulate it within an *ad-hoc* module,

:unify, coming with a general notion of `Unificator` – i.e. any algorithm aimed at computing MGU out of terms or clauses –, and a default implementation based on [MM82]. Developers may extend the default implementation by configuring e.g. when terms should be considered equal or not, or they can provide a different implementation for `Unificator`, in case they need a specific unification strategy, or, they prefer to adopt a different unification algorithm.

Another common need in LP is the in-memory storage of clauses into ordered – e.g. queues – or unordered – e.g. multisets – data structures, and their efficient retrieval via pattern-matching (e.g. unification). The *:theory* module follows this purpose, by providing notions such as `ClauseQueue`, `ClauseMultiset`—all coming both in an *immutable* (access-efficient) and *mutable* (update-efficient) implementation. These data structures differ from ordinary collections as they enable a unification-based retrieval and indexing of clauses. Prolog's notions of theory, and static/dynamic KB are built on top these data structures, exploiting the most adequate implementation in each case.

The practice of LP may also involve several ancillary operations over terms and clauses, e.g.: *(i)* formatting – into some *customisable* form –, *(ii)* (de)serialisation – into/from open data-representation formats such as JSON or YAML –, and *(iii)* parsing—out of a particular concrete syntax, such as, e.g., Prolog syntax. While formatting is considered a *:core* functionality, attained via the `TermFormatter` type, (de)serialisation and parsing come with their own modules. Thus, module *:serialize-core* (resp. *-theory*) supports the serialisation and deserialisation of terms (resp. theories) into JSON or YAML, according to a human-readable schema. This is aimed at supporting distributed applications needing to exchange logic knowledge over the network. Similarly, parsing terms (resp. theories) in Prolog syntax is currently supported through the *:parser-core* (resp. *-theory*) module, which is based on the well-known ANTLR technology [Par13] for language engineering.

A generic API for logic solvers is available as well within the *:solve* module. Conceptually, the purpose of this module is as simple as exposing the `Solver` type, which represents any entity capable of performing some sort of logic resolution to provide one or more `Solutions` to a logic query. However, resolution involves many practical aspects – such as errors management, extensibility via libraries, I/O, etc. – which are *orthogonal* w.r.t. any particular resolution strategy. This is why *:solve* is a quite articulated – despite not directly usable – module.

While developers may easily build their inference procedure of choice by providing an implementation for the `Solver` interface – possibly selectively reusing features from *:solve* –, two implementations are currently available as part of 2P-Kt – namely *:solve-classic* and *-streams* –, both implementing Prolog's SLDNF resolution strategy. In particular, *:solve-classic* is based on the work of Piancastelli et al. [PBOR08] and is currently stable, while ***:solve-streams*** is an experimental attempt of implementing Prolog via LP following the idea proposed in [Car84]. Notably, none of them relies on the Warren Abstract Machine [War83]—the computational model Prolog is commonly built upon.

A generic API for developing Prolog-like predicates in Kotlin is available as well. It heavily leverages FP and OOP to let developers extend their solvers with libraries of complex functionalities – possibly involving backtracking or side-effects – which are easier to implement in Kotlin than through LP. There, lazy streams of data are treated as flows of solutions to be enumerated via backtracking. This makes 2P-KT very well-suited for handling possibly infinite streams of data via LP – an essential feature in modern AI – as further discussed in chapter 10.

User experience (UX) is enabled by two more modules – namely, ***:repl*** and ***:ide*** –, which provide a CLI and GUI, respectively. While they both target JVM-specific UX, an experimental web-based GUI is available at [Pla21], targetting JS-specific UX.

The many ***:dsl-**** modules in fig. 9.1 are aimed at supporting the Kotlin-based DSL for LP described in [CCS+20]. It consists of façade to 2P-KT API aimed at blending the OOP, FP, and LP programming paradigms via Kotlin's flexible syntax. Within the 2P-KT project, this DSL is extensively exploited for unit testing.

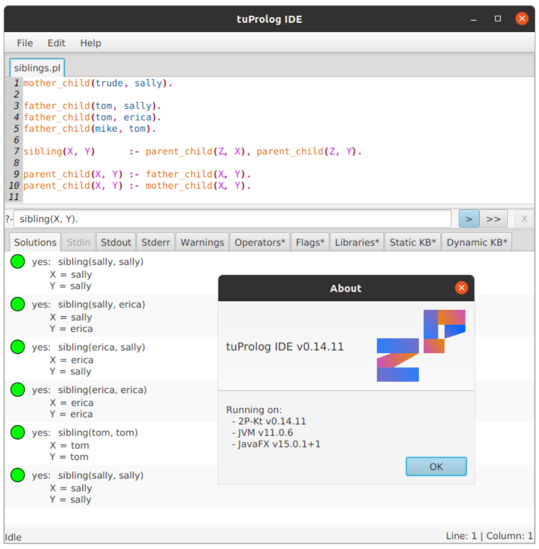
Finally, the ***:oop-lib*** module is experimental logic library aimed at supporting the exploitation of OOP from within logic programs.

9.3 Illustrative Examples

The 2P-KT GUI (fig. 9.3a) consists of a minimal IDE based on Java FX. It supports users willing to exploit LP interactively, possibly editing a theory repeatedly, performing different queries, and inspecting the mutable internals of the underlying solver. Accordingly, the GUI lets users open several files at once, perform queries one-by-one or all-at-once, or inspect the currently loaded libraries, operators, flags, etc. Syntax colouring completes the picture, easing users' writing of logic theories.

Conversely, the 2P-KT CLI (fig. 9.3b) lets users issue their queries against logic solvers via a textual console. It supports both an interactive and non-interactive operation mode. In the former case, the application consists of a Read-Eval-Print-Loop accepting logic queries from `stdin` and progressively prompting solutions to `stdout`. In the latter case, queries and theories are provided as arguments upon CLI launch and the program terminates after prompting all possible solutions.

The 2P-KT Playground (fig. 9.3c) is currently a proof-of-concept web application which only lets users write/load theories, issue logic queries, and visualise the


tuProlog IDE
File Edit Help
siblings.pl
Solutions Stdin Stdout Stderr Warnings Operators* Flags* Libraries* Static KB* Dynamic KB*
About
OK
Idle
Line: 1 | Column: 1

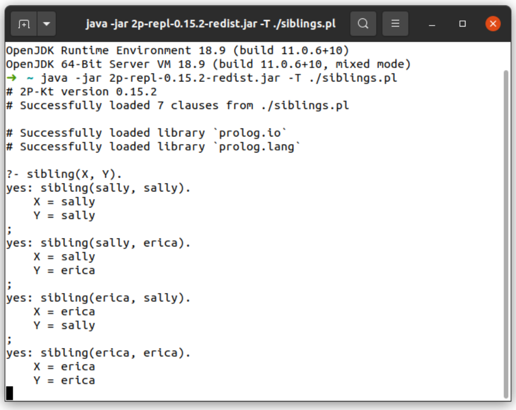
```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

```
?- sibling(X, Y).
```

```
yes: sibling(sally, sally)
     X = sally
     Y = sally
yes: sibling(sally, erica)
     X = sally
     Y = erica
yes: sibling(erica, sally)
     X = erica
     Y = sally
yes: sibling(erica, erica)
     X = erica
     Y = erica
yes: sibling(tom, tom)
     X = tom
     Y = tom
yes: sibling(sally, sally)
     X = sally
     Y = sally
```

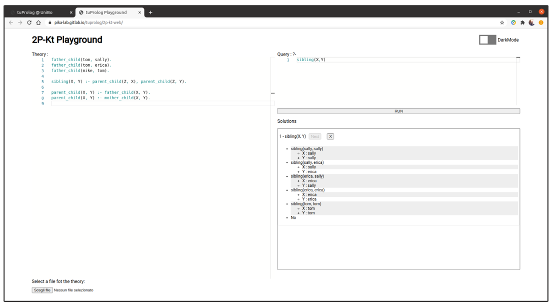
```
tuProlog IDE v0.14.11

Running on:
 - 2P-Kt v0.14.11
 - JVM v11.0.6
 - JavaFX v15.0.1+1
```

(a) The 2P-Kt GUI



```
OpenJDK Runtime Environment 18.9 (build 11.0.6+10)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.6+10, mixed mode)
➜ ~ java -jar 2p-repl-0.15.2-redist.jar -T ./siblings.pl
# 2P-Kt version 0.15.2
# Successfully loaded 7 clauses from ./siblings.pl

# Successfully loaded library `prolog.io`
# Successfully loaded library `prolog.lang`

?- sibling(X, Y).
yes: sibling(sally, sally).
    X = sally
    Y = sally
;
yes: sibling(sally, erica).
    X = sally
    Y = erica
;
yes: sibling(erica, sally).
    X = erica
    Y = sally
;
yes: sibling(erica, erica).
    X = erica
    Y = erica
```

(b) The 2P-Kt CLI

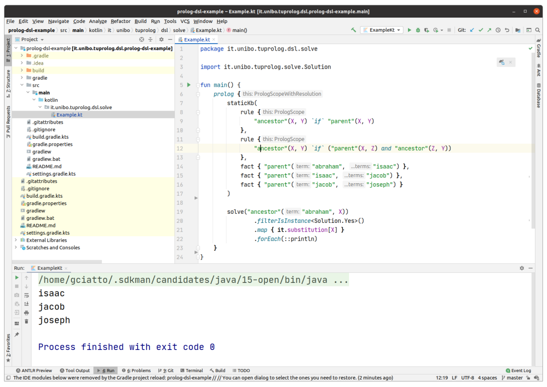

tuProlog @ UniBo
tuProlog Playground
pika-lab.gitlab.io/tuprolog/2p-kt-web/
2P-Kt Playground
DarkMode
Theory :
Query : ?-
RUN
Solutions
Next
X
Select a file fot the theory:
Scegli file Nessun file selezionato

```
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

```
sibling(X,Y)
```

```
1 - sibling(X, Y)
 • sibling(sally, sally)
    ◦ X : sally
    ◦ Y : sally
 • sibling(sally, erica)
    ◦ X : sally
    ◦ Y : erica
 • sibling(erica, sally)
    ◦ X : erica
    ◦ Y : sally
 • sibling(erica, erica)
    ◦ X : erica
    ◦ Y : erica
 • sibling(tom, tom)
    ◦ X : tom
    ◦ Y : tom
 • No
```

(c) The 2P-Kt Playground [Pla21]


prolog-dsl-example – Example.kt [it.unibo.tuprolog.dsl.prolog-dsl-example.main]
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
prolog-dsl-example src main kotlin it unibo tuprolog dsl solve Example.kt main()
Project
prolog-dsl-example [it.unibo.tuprolog.dsl.prolog-dsl-example]
.gradle
.idea
build
gradle
src
main
kotlin
it.unibo.tuprolog.dsl.solve
Example.kt
.gitattributes
.gitignore
build.gradle.kts
gradle.properties
gradlew
gradlew.bat
README.md
settings.gradle.kts
External Libraries
Scratches and Consoles
Run: ExampleKt
ANTLR Preview Tool Output 4: Run 6: Problems 9: Git Terminal Build TODO Event Log
The IDE modules below were removed by the Gradle project reload: prolog-dsl-example // // You can open dialog to select the ones you need to restore. (2 minutes ago)
12:19 LF UTF-8 4 spaces master

```kotlin
package it.unibo.tuprolog.dsl.solve

import it.unibo.tuprolog.solve.Solution

fun main() {
    prolog {
        staticKb(
            rule {
                "ancestor"(X, Y) `if` "parent"(X, Y)
            },
            rule {
                "ancestor"(X, Y) `if` ("parent"(X, Z) and "ancestor"(Z, Y))
            },
            fact { "parent"("abraham", "isaac") },
            fact { "parent"("isaac", "jacob") },
            fact { "parent"("jacob", "joseph") }
        )

        solve("ancestor"("abraham", X))
            .filterIsInstance<Solution.Yes>()
            .map { it.substitution[X] }
            .forEach(::println)
    }
}
```

```
/home/gciatto/.sdkman/candidates/java/15-open/bin/java ...
isaac
jacob
joseph

Process finished with exit code 0
```

(d) Usage of the Kotlin DSL for Prolog [CCS+20], within an IDE

Figure 9.3: Usage examples for 2P-Kt

corresponding solutions in their browsers. The key point here is that a full-fledged 2P-KT application can be executed in-browser in a server-less fashion. In fact, our Playground only requires Internet connection upon page loading. After that, it does not interact with the server any longer as the loaded JS scripts are more than sufficient to make 2P-KT usable from within users' browsers. For this reason, logic computations performed through our Playground need not any sandbox – as no server is needing to be protected against DoS attacks –, nor logic solvers need API limitation for security reasons.

Finally, 2P-KT's DSL for Prolog can be exploited within Kotlin projects as shown in fig. 9.3d. Essentially, it provides a syntactical way to inject LP into Kotlin scripts. To make this possible, users must import one or more *`:dsl-*`* module as dependencies into their Kotlin projects, e.g. via Gradle or any other build system. As discussed in [CCS+20], the adoption of this DSL makes the life of logic programmers easier, as they can automatically inherit the many tools available for Kotlin development—e.g. type checking, linters, code completion, debugging, etc.

9.4 Impact

We expect 2P-KT technology to have an impact on many research areas.

Within the scope of LP, for instance, 2P-KT provides a well-grounded technological basis for implementing (or building variants and extensions of) the many solutions proposed into the literature—there including abductive inference procedures [FK97], rule induction methods [Md94], probabilistic reasoning [dK15], labelled LP [CDDO18], etc.)

Furthermore, as recently shown in [CCMO21a], the multi-agent systems community have quite an appetite for *interoperable* and general-purpose logic-based technologies. There, 2P-KT may provide a technological substrate supporting agent automated reasoning via manifold reasoning mechanisms.

Similarly, 2P-KT represents a valuable technological choice within the field of Coordination [MC94]. As demonstrated in [CDMSL+20], many tuple-based coordination models and technologies rely on LP and logic-based technologies at the fundamental level. There, 2P-KT enables the implementation of *interoperable* LINDA tuple spaces – such as in TUSOW [CROM19] – or tuple *centres*—as we plan to do in TuCSoN [OZ99].

Moreover, as discussed in [CCS+20], 2P-KT impacts on programming paradigms as well. In fact, while most successful programming paradigms (imperative programming, OOP, FP) are being increasingly *blended* into modern programming languages, LP remains somewhat isolated. In this context, 2P-KT's DSL for LP paves the way towards the integration of LP with other programming paradigms.

Finally, we expect 2P-KT will have a role to play in the field of explainable

AI. There, the integration among symbolic and sub-symbolic AI is considered a strategical research direction [CCO20] and 2P-Kt may offer a sound technological basis to this purpose.

2P-Kt adoption. While tuProlog has been exploited both in the industry and in the academia[3], 2P-Kt has been used in the academia only—so far. However, it has already worked – or, will work in the near future – as the technological basis of many scientific contributions. Some, such as TuCSoN [OZ99], ReSpecT [OD01], LPaaS [CCM+18], or Tenderfone [CMOZ20] leveraged on tuProlog for their implementation, and are now being migrated on 2P-Kt. Others, such as TuSoW [CROM19], Arg2P [PCOS20], or the Kotlin-based DSL for LP proposed in [CCS+20] are already based on 2P-Kt.

Research directions stemming from 2P-Kt. Currently, 2P-Kt is already enabling the exploration of many interesting research questions thanks to its modularity and interoperability. Any research line involving symbolic manipulation or automated reasoning is then likely to benefit from 2P-Kt functionalities. Generally speaking, 2P-Kt paves the way towards: the coexistence and integration of different LP aspects, the hybridisation of LP with other AI techniques, and the exploitation of LP in building flexible intelligent systems. Along these lines, our goals involve: *(i)* the creation of comprehensive solvers capable of exploiting multiple inference procedures, knowledge-representation means, etc. at once in answering users' queries, *(ii)* the construction of hybrid systems where logic programmers can transparently exploit machine learning (ML) and sub-symbolic AI, and *(iii)* the injection of LP into cognitive agents architectures.

So, as far as goal *(i)* is concerned, we are currently exploring the design of probabilistic, abductive, or concurrent resolution under a unique API. These functionalities may constitute novel modules enriching the 2P-Kt ecosystem. This would enable further research towards, e.g., *mixed* reasoning processes, where multiple inference procedures can be dynamically interleaved while answering some user's query.

As far as goal *(ii)* is concerned, we are currently exploring the design of a logic-based API for ML and, in particular, neural networks. The API allows logic-programmers to define, train, assess, and use sub-symbolic predictors via LP. At the technical level, the API are reified into yet another module enriching the 2P-Kt ecosystem. This would enable further research towards, e.g., the integration of symbolic and sub-symbolic AI, the automation of ML workflows, and the exploitation of fuzzy knowledge induced from data in LP.

[3]`http://apice.unibo.it/xwiki/bin/view/Tuprolog/Users`

Finally, as far as goal *(iii)* is concerned, we are currently exploring the integration of multiple logics within BDI architectures. The idea here is to enable intelligent agents to perform the most adequate sort of reasoning or knowledge-representation means for the situation at hand. This would enable further research towards, e.g., the exploitation of different logics (e.g. first-order, temporal, spatial, deontic, etc.) to support intelligent, context-specific behaviour for software agents. There, 2P-Kt provides a common ground for the implementation of automated reasoners supporting each logic.

Accordingly, in the reminder of this thesis, we explore a number of contributions stemming from (and enabled by) 2P-Kt.

In particular, in chapter 10, we introduce a fundamental mechanism – namely, *primitives* – which 2P-Kt solvers may exploit to lazily handle streams of data. Despite its simplicity, the mechanism is very powerful, as it enables the interoperability of LP with other runtimes, e.g. JVM libraries. This, in turn, enables most of the subsequent contributions discussed in this thesis. Similarly, in chapter 11, we discuss the integration of logic, object-oriented, and functional programming at the paradigm and language level, as made possible by 2P-Kt.

In chapter 12, we propose the design of a logic API for ML, aimed at enabling the manipulation of ML predictors in LP, and, therefore, the creation of hybrid (symbolic + sub-symbolic) AI solution. We also discuss how such API can be reified into yet another module expanding the 2P-Kt ecosystem.

In chapter 13, we propose the design and implementation of a "platform for symbolic knowledge extraction" (PSyKE) reifying the vision and the architecture discussed in section 6.3. The proposed technology enables the extraction of logic rules out of sub-symbolic predictors. Such rules are of course represented as clauses, as enabled by 2P-Kt.

Furthermore, in chapter 14 we discuss the extension of the 2P-Kt ecosystem towards probabilistic logic programming. Despite the benefits of supporting probabilistic reasoning, such extension confirms the advantages of the ecosystem-based approach. In fact, we show how probabilistic reasoning support can be achieved by reusing as much functionalities from the 2P-Kt ecosystem as possible.

Finally, in chapter 15 we discuss a number of future research directions, stemming from or overlapping 2P-Kt. Despite some steps have already been performed along these directions, they are not mature enough to be included as chapters in this thesis. Hence, we report them as future works, providing insights about their backgrounds and presenting what we have already done so far.

Chapter 10

Bridging LP and Stream Processing

This chapter contains contributions from the following works of ours: [CCO21b]

Streams are a powerful abstraction in computer science as they enable the processing of huge amounts of data, especially when keeping all data in memory would be impractical or infeasible. In the era of the Internet of Things (IoT) and data-driven artificial intelligence (AI), the ability of manipulating possibly unlimited streams of data is a must-have for all programming paradigms and languages. Indeed, a growing amount of application scenarios are characterised by the pervasive exploitation of smart devices generating/capturing huge amounts of data, as well as of the software infrastructures aimed at processing them.

A stream is an *ordered sequence* of data that may or may not be limited in length. Stream processing facilities are thus commonly constructed in such a way that streams are *lazily consumed*, in order to minimise the amount of required memory—which may be soon saturated otherwise. However, despite all sorts of streams are lazily consumed, categories may be drawn depending on how they are *generated*. Depending on how the are *generated*, streams are either *cold* (a.k.a. *pull*) or *hot* (a.k.a. *push*). Each item of a cold stream is generated on the fly, as soon as a consumer *pulls* it from the stream. In the case of hot streams, instead, an external entity is supposed to be in charge of generating items and *pushing* them to the stream, so that consumers can retrieve them in a FIFO way.

Cold streams are the simplest ones. A cold stream can be naturally attained via functional programming and higher-order functions (e.g. `map`, `filter`, `reduce`): this is why mainstream programming languages such as Java, C#, Python, JavaScript, Scala, Kotlin, etc., are being extended to blend functional features and constructs for dealing with streams. Conversely, hot streams are more complex, as they require data to be buffered while waiting for consumption—making them ideal for *temporally* decoupling data consumers and producers. In particular, hot streams are key enablers of advanced stream processing techniques, such as sliding windows, or complex event processing (CEP)—which are deeply entangled with the *time*-related aspects of data production.

In this scenario, logic programming (LP), as well, has its role to play, both in data-driven AI – in particular in relation to explainable systems [CCO20] – or in the IoT [CCM$^+$18]. For instance, LP and rule-based frameworks are generally recognised as well-suited to support CEP [AFR$^+$10, ARFS12], as they are expressive enough to capture complex events from hot streams. Similarly, answer-set programming (ASP) has been extensively exploited as a means for reasoning over hot streams of data [EIST05, BEF17, BDTE18].

In this chapter, we focus on the Prolog [CR93] programming language—arguably, the most popular LP language. Currently, Prolog can hardly be considered as a suitable stream-processing technology [TWS19], as it provides minimal support for consuming both cold and hot streams. However, we believe that this should be reconsidered because Prolog already supports the lazy exploration of possibly infinite search spaces via *backtracking*. Thus, the problem with Prolog is not to discuss *whether* it supports stream processing or not, but rather *how*.

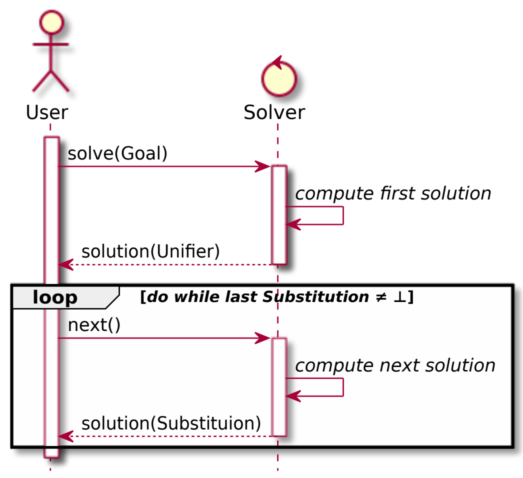
Existing solutions extend Prolog with syntactical, semantical, or library enhancements aimed at supporting cold streams explicitly. Conversely, in this chapter, we discuss how Prolog can be reinterpreted as a *stream processing tool*, capable of manipulating both cold and hot streams of data. In particular, our solution does not affect the syntax (nor the operation) of the Prolog language. More precisely, we show how Prolog predicates may be interpreted as *primitives* of streams to be lazily consumed via backtracking. Along this line, we present an abstract design for Prolog solvers based on finite-state machines, aimed at supporting our notion of primitives. Finally, a practical demonstration based on the 2P-Kt technology [2P-21] is discussed showing how primitives may let a Prolog solver consume events from the external world in a transparent way.

10.1 Logic Solvers as Streams Prosumers

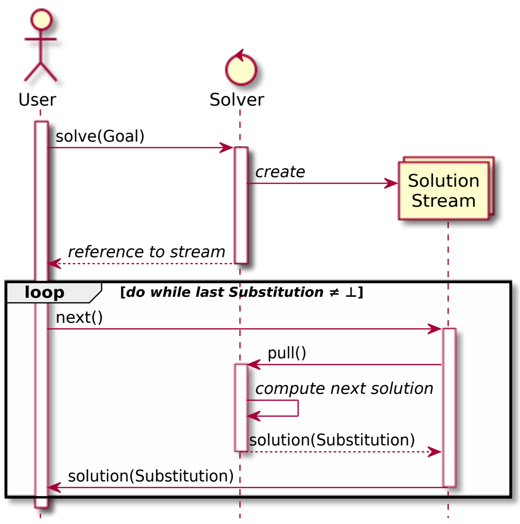
10.1.1 Logic solvers as stream producers

Logic solvers *à la Prolog* are typically queried *interactively* by LP users in different *modes*, which are naturally captured by the message passing perspective adopted in fig. 10.1. The most common mode of interaction among users and logic solvers is summarised in fig. 10.1a: users submit *queries* (a.k.a. *goals*) to a logic solver – e.g. a Prolog interpreter – via some *ad-hoc* operation—e.g., `solve`. Assuming that one or more *solutions* exist, the solver computes and returns one of them—typically in terms of a *unifying substitution*, assigning values to the query variables of interest

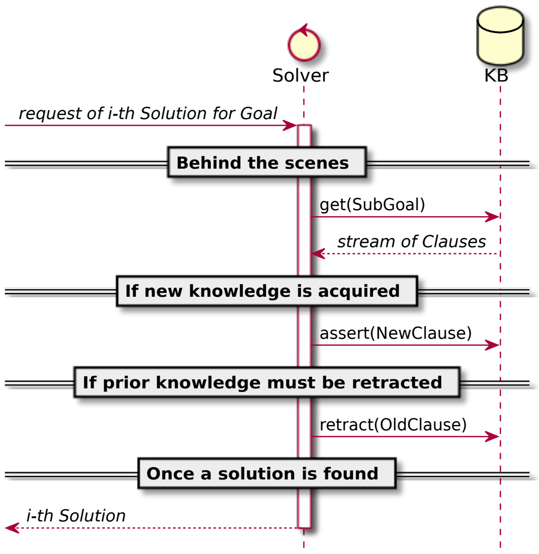
Figure 10.1: Interaction modes between logic solvers and users or KB



(a) In *stateful interaction* mode, solvers expose two functionalities: `solve` – to compute the first solution to some query –, and `next`— to compute subsequent solutions to the same query



(b) In *stream-oriented interaction* mode, solvers expose one functionality: `solve`— which simply returns a stream of solutions that users may lazily consume



(c) Interaction among a logic solver and its KB. As possibly mutable containers of knowledge KB expose three main functionalities: `get`(C) returns a *stream* of clauses matching C via unification, whereas `assert`(C) (resp. `retract`(C)) adds (resp. removes) some clause (unifying with) C

for the user. However, the user may be interested in solutions other than the first one: so, the solver should expose one further operation – e.g., `next` – letting users asking for further solutions to some previously-submitted query. Finally, when no (more) solutions are available for a query, the solver can return one (last) answer carrying the *failed* substitution (represented by $\perp$ in fig. 10.1) instead of a unifier.

This mode of interaction is very effective since it enables the *lazy* enumeration of a possibly infinite amount of solutions. However, it comes with a few drawbacks. First, despite logic solvers are actually capable of generating streams of solutions, the notion of stream is somewhat *implicit* in the solver machinery—therefore, not explicitly exploitable. Second, solvers are *stateful*, in that they are responsible to keep track of the status of the interaction with each querying user.

To overcome these issues we suggest a shift of perspective, as depicted in fig. 10.1b. There, users and solvers interact in a *stream-oriented* mode, where the stream of solutions is *explicit* and the interaction between solvers and users is *stateless*. Thus, solvers expose just one operation – i.e., `solve` – accepting a user's query and returning a reference to the related *cold* stream of solutions. Users just need solvers to create solution streams that users can then lazily consume on demand. Of course, solutions can still be produced lazily behind the scenes: whenever a user tries to consume a new solution, it can be computed on the fly.

Thus, even though interaction does not change from the operational viewpoint, our approach overcomes the limits of traditional logic solvers: solution streams here are *explicitly* represented, and can therefore be *manipulated* as such.

10.1.2 Logic solvers as stream consumers

By adopting a message passing perspective, logic solvers do not interact with users only. Indeed, logic solvers typically act on a *knowledge base* (KB). In the general case, KBs are containers of the specific knowledge required by solvers to compute solutions to users' queries. For instance, KB for Prolog solvers contain both rules and facts as Horn clauses, and are either static or dynamic.

From an interaction perspective, however, a KB is just a component exploited by solvers as part of their resolution process. More precisely, solvers may need KB to retrieve some clauses, selected via unification, or, to retract or store some knowledge possibly learned/acquired during the resolution.

In particular, clause retrieval highlights how the interaction between solver and KB can be described in terms of streams as well. As depicted in fig. 10.1c, clause retrieval from KB can be modelled as an operation – e.g., `get` – accepting a clause template C and returning the *stream* of clauses unifying with C currently stored into the KB. The solver can then consume the stream as needed, e.g. either lazily or not, depending on the search strategy adopted.

Finally, storing a clause in the KB can be modelled as an `assert` accepting a clause C and adding it to the KB, whereas clause retraction can be modelled as a `retract` accepting a clause template C and removing a clause C' unifying with C. Both operations could be exploited either by the solver or by some *external* entity willing to affect the solver's knowledge.

10.1.3 Solvers vs. the World

Yet, how can logic solvers deal with event streams coming from the external world? Once KBs are recognised as individual entities, a trivial answer could be: *via KB*. External events may indeed be *reified* into actual knowledge to be stored into some solver's KB. In this scenario, external event streams should be translated into a sequence of `assert`ions aimed at injecting events into the KB, as facts. The solver could then lazily consume the events by `get`ting or `retract`ing the corresponding facts from the KB.

There are, however, two major drawbacks in this approach. First, the reification of events into KB requires space. Second, solvers do not necessarily have to process or *consume* reified events—thus a lot of space is wasted. Accordingly, a different approach is required to let solvers consume event streams from the external world without reifying them unnecessarily.

In this work, we propose *primitives* as the basic means to let solvers interact with the external world. A primitive is a special Prolog facility capable of affecting and inspecting the external world via some I/O facility (fig. 10.2). It is invoked by a solver and produces a *stream of facts* to be consumed by the same solver. However, from the solvers' perspective, primitives are ordinary built-in predicates denoted by *signatures*—i.e., name/arity couples of the form p/n.

More precisely, whenever the solver needs to compute the assignment of variables T_i satisfying relation $p(T_1, \ldots, T_n)$, it can trigger the primitive denoted by p/n (if it exists), by sending the p/n primitive a *request* providing a snapshot of the current resolution context and possibly an initial assignment of some T_i. The primitive answers by providing a stream of *responses* – each one with some possible complete assignment of T_i – that the solver can consume accordingly to its resolution strategy—i.e., possibly later. To produce responses, primitives may take into account several information sources – e.g., the resolution context, the external world – as a part of the request. They may also attempt to *affect* the external world via some I/O *action*—e.g., triggering a sensor.

Depending on the numbers of responses a primitive provides, it can either be classified as either *functional* or *relational*. Functional primitives produce just one response and their execution is therefore analogous to the execution of a function, as they consume an input and return a single result. Conversely, relational primitives produce two or more responses.

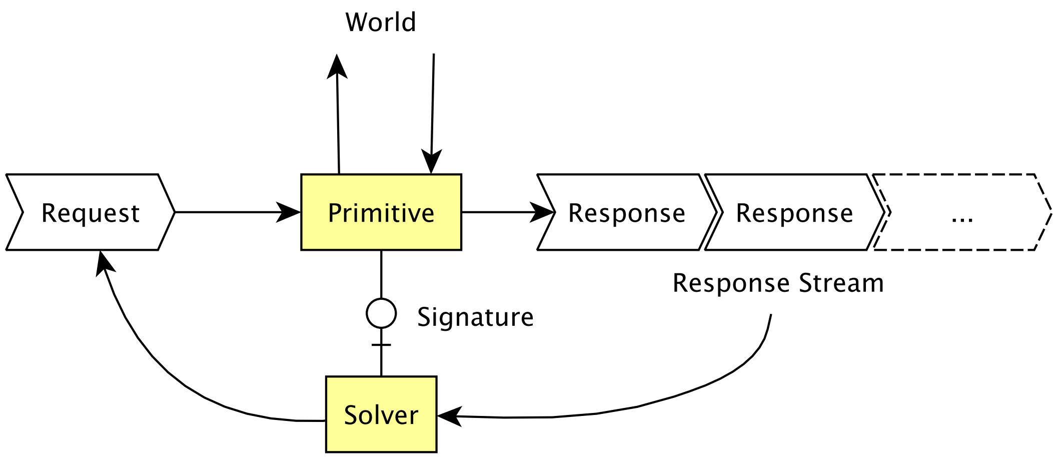


Figure 10.2: Dataflow and component view of *primitives*, i.e. solvers' gates towards the external world

10.1.4 Example: TSP in Prolog

Let us consider for instance the case of a user exploiting a standard Prolog system to solve arbitrary instances of the Traveling Salesman Problem (TSP).

Let us assume the system requires maps to be represented as facts in the form `path(+Src, +Dst, +Cst)` – each one representing an undirected path between two locations, and the estimated cost –, like e.g.:

```
path(bucarest, giorgiu, 90).
path(bucarest, pitesti, 101).
path(pitesti, 'rimnicu vilcea', 97).
path(pitesti, craiova, 138).
path('rimnicu vilcea', craiova, 146).
...
```

Under this assumption, Prolog exposes a predicate `tsp(?Cities, ?Circuit, ?Cost)` aimed at computing the best `Circuit` for some set of `Cities`, and the corresponding `Cost`—where, `Cities` is a set of cities, `Circuit` is a list of cities to be visited in a row, and `Cost` is an integer. Following a purely-logical interpretation, the predicate represents a ternary relation `tsp` $\subseteq 2^{\mathcal{C}} \times \mathcal{C}^* \times \mathbb{N}$ grouping subsets of cities, lists of cities, and non-negative integers, where $\mathcal{C}$ is the set of all cities mentioned in the KB as either the first or second argument of a `path/3` fact, and $\mathcal{C}^*$ is the Kleene-closure of $\mathcal{C}$. Thus, an assignment of the `Cities`, `Circuit`, and `Cost` variables satisfies the predicate if

- `Circuit` $\equiv [c_0, \ldots, c_{n-1}, c_0]$, and
- `Cities` $\equiv \bigcup_{i=0}^{n-1} \{c_i\}$, and

- $\forall i \in \{1, \ldots, n\}$ `path(`c_{i-1}`, `$c_{i \bmod n}$`, `x_i`)` $\in$ KB, and
- `Cost` $\equiv \sum_{i=1}^{n} x_i$, and
- `Cost` is *minimal.*

Accordingly, because of Prolog backtracking, a query of the form:

```
?- tsp(Cities, Circuit, Cost).
```

would enumerate all minimally-costly circuits of all possible subsets of cities in $\mathcal{C}$, and their costs—one for each solution. Users may partially instantiate some variable in order to contextualise their queries: for instance, a query of the form:

```
?- tsp({pitesti, craiova, 'rimnicu vilcea'}, [pitesti | Others], Cost).
```

would enumerate all minimally-costly circuits starting in Pitesti, and involving the cities Craiova, and Rimnicu Vilcea.

The predicate `tsp/3` could be implemented declaratively in Prolog. In its simplest formulation, the predicate may leverage Prolog's depth-first strategy, and its backtracking mechanism to lazily generate all the possible circuits and select the less costly one: not likely the best possible strategy, yet a working one. However, better strategies have been proposed in the literature for solving the TSP, with efficient implementations built upon them—rarely based on pure Prolog. Here, instead, primitives make it possible to exploit external libraries for solving the TSP in Prolog as if they were implemented via LP.

For instance, we assume that a "ACME TSP" C library exists that solves TSP efficiently, which can be wrapped within a relational primitive `tsp/3` to be exploited by a Prolog solver. The primitive `tsp/3` should work as follows:

1. whenever the Prolog solver encounters a `tsp(Cities, Circuit, Cost)` sub-goal, it triggers the primitive via a *request* containing a snapshot of the current KB and the *actual* values of `Cities`, `Circuit`, and `Cost`;
2. the primitive reads *(i)* the map graph from the KB snapshot, and *(ii)* the cities from the actual value of `Cities`;
3. the primitive generates the stream of all the possible subsets of $\mathcal{C}$ and selects the ones unifying with the actual value of `Cities`, thus: if `Cities` is bound to a particular sub-set of cities, then the stream has just one element, otherwise it may have several ones;
4. for each sub-set of cities in the stream, the primitive triggers ACME TSP and computes the corresponding TSP solution, if any;

5. every time it is triggered, ACME TSP computes zero or more solutions for the TSP and returns them to the primitive;

6. for each TSP solution of each selected instantiation of `Cities`, the primitive yields a response to the solver;

7. each response may either contain a unifier – assigning `Cities` to the selected list of cities, `Circuit` to the minimally-costly circuit for those cities, and `Cost` to the cost of that circuit – or a failed substitution—informing the solver the `tsp/3` predicate should fail;

8. the solver can consume the response stream lazily via backtracking.

In other words, primitives can be exploited as a means to wrap external data producers and let the solver consume the data they produce via streams. In Prolog, streams of this sort are lazily consumed via ordinary backtracking: the solver lazily generates a new choice point for each element in the stream and handles them as usual. Solvers of different sorts may consume the stream differently—e.g. buffering (some slice of) it, or, handling each datum concurrently.

10.2 Solvers as Streams Prosumers via State Machine

In order to design a Prolog solver supporting our notion of primitive, we enhance the Prolog state machine proposed in [PBOR08] with the capability of lazily consuming streams of data coming from either a primitive or the KB (fig. 10.3). In particular, we change how the state machine manages the resolution of (sub-)goals, by supporting the selection of a primitive as a means to provide one or more solutions for (sub-)goals, other than the ordinary selection of rules from the KB.

The state machine in fig. 10.3 stems from the acknowledgement that a Prolog solver may solve a (sub-)goal by either selecting a primitive or a number of logic rules from the KB. In both cases, a stream of data must be lazily consumed by the solver—either carrying primitive responses or clauses from the KB.

Whenever a stream of data needs to be processed, there are essentially two major phases: the *opening* of the stream – where a channel between the stream producer and its consumer is created –, and the *consumption* of the stream—where items from the stream are sequentially processed. To support both phases, two more locations are included – namely Primitive Selection and Primitive Execution – respectively aimed at triggering a primitive and consuming the response stream it provides. Furthermore, to support a stream-oriented interaction among the solver and its KB, we model rule management as well through two locations, namely

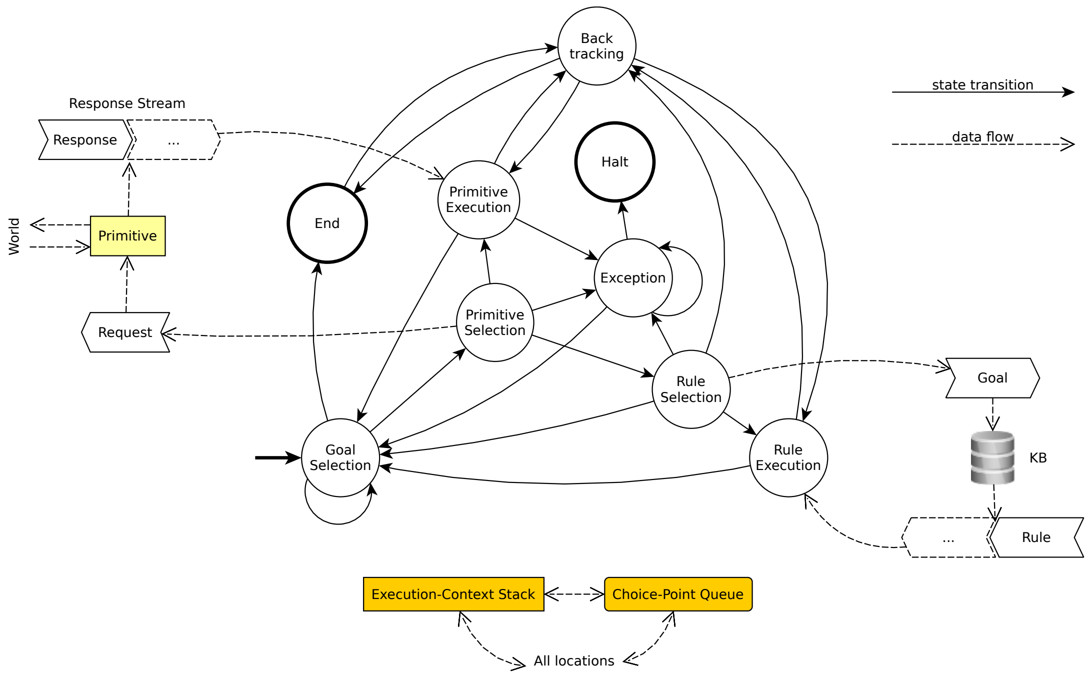


Figure 10.3: The primitive-enabled state machine governing Prolog solvers' behaviour. Location Goal Selection is the initial one, whereas End and Halt are the final ones. Location Primitive (resp. Rule) Selection is where primitives (resp. KB) are triggered (resp. queried) and their data streams are opened. Conversely, location Primitive (resp. Rule) Execution is where response (resp. clause) streams are lazily consumed

Primitive Selection and Primitive Execution, respectively aimed at querying the KB, and consuming the rule stream it provides.

All the other aspects are handled in the same way as in [PBOR08]. Thus, state machine execution is triggered whenever a user submits a query to the solver: when this is the case, execution starts from the Goal Selection location. Then, it may go through any location until it eventually reaches some *final* one (End or Halt), where a new solution is yielded—which the user can eventually consume. Once a solution is consumed, the user can either submit a new query or ask for the next solution. In the former case, the automaton is reset to the Goal Selection location. Conversely, the latter case is only possible if the last solution was provided by the End location. In that case, the automaton backtracks and looks for the next solution. This may involve stepping through Backtracking, then moving back into the Primitive (resp. Rule) Execution, in order to consume one more element from some previously-opened response (resp. clause) stream.

Overall, our state machine affects the operation of a Prolog solver as follows:

1. [Primitive Selection] whenever a new sub-goal is selected, the solver looks for a primitive whose signature matches the sub-goal one;

2. [Primitive Execution] if some are found, the solver considers the first response in the stream as a solution to the goal, and generates choice points for subsequent responses;

3. [Rule Selection] otherwise, if no primitive is selected for the current sub-goal, some rule is looked for instead, whose head unifies with the sub-goal;

4. [Rule Execution] if any such rule is found, resolution can proceed by addressing the rule's body as the next goal to be proved;

5. [Backtracking] otherwise, if no rule is found, the sub-goal is considered failed and resolution must backtrack.

Location Exception completes the picture by intercepting exceptions – possibly thrown by primitives as part of some response of theirs –, via the standard `catch /3` predicate.

Ordinary Prolog built-in primitives naturally fit the picture as they are re-interpreted as primitives by solvers. For instance, the `is/2` predicate can be considered a functional primitive accepting a variable and an expression and returning a single response assigning the variable to the value attained by reducing the expression – if possible –, or an exception—in case the expression cannot be reduced. Conversely, the `member/2` predicate can be considered as a relational primitive, enumerating all the possible items in a list. Accordingly, the aforementioned Primitive Selection location is where built-in primitives are selected for execution in place of rules from the KB.

10.2.1 Formal Description

In the reminder of this section we formally define the behaviour of a primitive-enabled state machine for Prolog. The discussion starts by introducing the basic notations, functions, and data structures aimed at formally describing the semantics of our state machine. Semantics is then provided via labelled transition systems.

Syntax and Notational Conventions

Here we recap the fundamental syntactical definitions our discussion relies upon. Most definitions here rely on the theoretical background provided in section 3.1.

Knowledge Bases. Let $\mathcal{A}$ be the set of all atomic logic formulæ, and let $\mathcal{H}$ be the set of all well-formed Horn clauses in the form:

$$a \leftarrow a_1, \ldots, a_n \quad \text{s.t. } n > 0$$

(a.k.a. rules) where $a, a_1, \ldots, a_n \in \mathcal{A}$ are logic predicates of arbitrary arity. Let us enumerate rules in $\mathcal{H}$ by h. Thus, we define knowledge bases as ordered containers of rules of the form $[h_1, h_2, \ldots]$.

Let K denote the knowledge base, with $K \in \mathcal{H}^*$, where $\mathcal{H}^*$ is the set of all possible KB.

Finally, let $get : \mathcal{H}^* \times \mathcal{H} \rightarrow \mathcal{H}^*$ be a function defined as follows:

$$get(K, h) = \begin{cases} [h' \mid get(K')] & \text{if } K \equiv [h' \mid K'] \wedge mgu(h, h') \neq \bot \\ [get(K')] & \text{if } K \equiv [h' \mid K'] \wedge mgu(h, h') = \bot \\ [] & \text{if } K \equiv [] \end{cases}$$

aimed to select all rules in K unifying with the clause h.

Primitives. Let $\mathcal{F}$ be the set of all predicate symbols, enumerated by f, and let $\mathbb{N}$ be the set of natural numbers. Thus,

- let $\mathcal{F} \times \mathbb{N}$ denote the set of all possible signatures and
- let $(f, n) \in \mathcal{F} \times \mathbb{N}$ denote the generic signature of a n-ary predicate whose functor is f.

Let us define the function $signature : \mathcal{A} \rightarrow \mathcal{F} \times \mathbb{N}$ as:

$$signature(f(t_1, \ldots, t_n)) = (f, n)$$

that computes the signature of any possible atomic predicate.

Let Θ be the set of all possible substitutions – including the failed substitution $\bot$, the empty unifier $\varnothing$, and any unifier in the form $\{V_1 \mapsto t_1, V_2 \mapsto t_2, \ldots\}$ where V_i are logic variables and t_i are logic terms – and let us enumerate elements in Θ by θ.

Let then $\mathcal{X}$ be the set of all possible *exceptions* – i.e., arbitrary terms describing error situations –, enumerated by x.

Accordingly, $\mathcal{R} = \mathcal{X} \cup \Theta$ represents the set of all possible primitive *responses*, enumerated by r.

Then, let $\mathcal{P}$ be the set of all possible primitives with the form:

$$p : \mathcal{H}^* \times \mathcal{A} \to \mathcal{R}^*$$

In other words, we call "primitive" any function $p \in \mathcal{P}$ accepting a knowledge base $K \in \mathcal{H}^*$ and a goal $a \in \mathcal{A}$ as input and producing a stream of responses $p(K, a) = \bar{r} \in \mathcal{R}^*$ as output—where each response $r \in \bar{r}$ may either be a substitution or an exception.

Finally, we define a primitive store I as a relation of the form:

$$I \subseteq \mathcal{F} \times \mathbb{N} \times \mathcal{P}$$

that is, any possible indexing of primitives by signatures. Given a particular primitive store I, we enumerate its elements by (f, n, p).

Solver Automaton. We define an *execution context* E as a tuple of the form $\big(\theta, \bar{a}, \bar{h}, \bar{p}\big)$ where:

- $\theta \in \Theta$ is a substitution;
- $\bar{a} \in \mathcal{A}^*$ is a stream of *goals*;
- $\bar{h} \in \mathcal{H}^*$ is a stream of *rules*;
- $\bar{r} \in \mathcal{R}^*$ is a stream of primitive *responses.*

Accordingly, letting $\mathcal{E}$ denote set of all possible execution contexts, we define an execution-context *stack* $\mathbf{E}$ as a list of the form $[E_0, E_1, \ldots] \in \mathcal{E}^*$, where E_0 can either be called the "*current* execution context" or the "*top* of the execution-context stack".

Conversely, we define a *choice point* C as any item in $\mathcal{E}^*$—i.e., a snapshot of a *whole* execution-context stack. We then define a choice-point *queue* $\mathbf{C}$ as a list of the form $[C_0, C_1, \ldots]$, and we denote by $\mathcal{C}^*$ the set of all possible choice-point queues. There, C_0 can either be called the "*next* choice point" or the "*head* of the choice-point queue". Let *append* : $\mathcal{C}^* \times \mathcal{E}^* \to \mathcal{C}^*$ be a function defined as:

$$append(\mathbf{C}, C) = \begin{cases} [C] & \text{if } \mathbf{C} \equiv [] \\ [C_1, \ldots, C_n, C] & \text{if } \mathbf{C} \equiv [C_1, \ldots, C_n] \end{cases}$$

serving the purpose of appending a new choice point at the end of a choice-point queue.

Finally, let $\mathcal{L}$ denote the set containing all the locations depicted in fig. 10.3:

$$\mathcal{L} = \{\mathsf{Goal\ Selection}, \mathsf{Primitive\ Selection}, \mathsf{Primitive\ Execution}, \mathsf{Rule\ Selection}, \\ \mathsf{Rule\ Execution}, \mathsf{Backtracking}, \mathsf{Exception}, \mathsf{End}, \mathsf{Halt}\}$$

We enumerate items in $\mathcal{L}$ by L.

Semantics

The semantics of our Prolog state machine can be described in terms of *states* and *transitions* between them. In this regard, a *state* is a tuple of the form $\langle L, \mathbf{E}, \mathbf{C}\rangle$ where

- $L \in \mathcal{L}$ is a location,
- $\mathbf{E} \in \mathcal{E}^*$ is an execution-context stack,
- $\mathbf{C} \in \mathcal{C}^*$ is a choice-point queue.

Accordingly, the state machine semantics is defined as a *labelled transition system* $\langle \mathcal{S}, \Lambda, s_0, \longrightarrow\rangle$ where:

- $\mathcal{S}$ is the set of all possible states,
- $\Lambda = \{\tau\} \cup \mathcal{X} \cup \Theta$ is a set of labels,
- $s_0 \in \mathcal{S}$ is the initial state,
- $\longrightarrow \subseteq \mathcal{S} \times \Lambda \times \mathcal{S}$ is a transition relation dictating how state may evolve in time.

There, transition labels in Λ denote relevant observable events, while the anonymous label τ denotes internal events. Observable events of interest can be, for instance, the production of either a positive or negative solution – denoted by the corresponding substitution in Θ – or the production of an exceptional solution—denoted by the corresponding exception in $\mathcal{X}$. In the following, for the sake of notation simplicity, we write $s \xrightarrow{\lambda} s'$ instead of (s, λ, s') to refer to transitions in $\longrightarrow$.

Assuming that a KB K and a primitive store I are provided, and that the initial state $s_0 \in \mathcal{K}$ is always a tuple of the form:

$$\langle \textsf{Goal Selection},\ [(\varnothing, [g_0], [], [])],\ []\rangle$$

for some initial goal g_0 – meaning that *(i)* the initial location is always Goal Selection, *(ii)* the current context initally only contains the empty unifier $\varnothing$ and g_0, and *(iii)* the choice-point queue is initially empty – we can *intensionally* define the admissible transitions in $\longrightarrow$ via the following transition rules—each one corresponding to an arrow in fig. 10.3.

Goal Selection. The purpose of the Goal Selection location is to decide what to do next depending on which and how many (sub-)goals are in the current execution context. There are three relevant situations handled in this location, by as many transition rules. More precisely, if the current execution context does not contain any goals, this implies either that a new solution should be yielded, or the top execution context should be popped from the stack. Conversely, if the current execution context does hold at least one goal, the automaton commits to that goal—meaning that it tries to prove its truth via subsequent transitions.

Accordingly, the following transition rule handles the case where there is only one last execution context on the stack with no more goals—thus, implying the automaton should move into the End location and a new solution should be yielded:

$$\frac{E = (\theta, [], [], [])}{\langle \textsf{Goal Selection}, [E], \mathbf{C}\rangle \xrightarrow{\theta} \langle \textsf{End}, [E], \mathbf{C}\rangle}$$

The positive or negative solution depends on the θ substitution of the last execution context E, which can be either a unifier or the failed substitution $\bot$. In the former case, θ synthesises all the variable assignments computed so far by the automaton.

Conversely, the following transition rule handles the case where the current execution context has no more goals, but the stack contains more execution contexts. When this is the case, the automaton simply pops the current execution context from the stack and holds the Goal Selection location:

$$\frac{E_0 = (\theta_0, [], [], []) \qquad E_1 = (\theta_0, \bar{g}, [], []) \qquad E_1' = (\theta_1, \bar{g}, [], [])}{\langle \textsf{Goal Selection}, [E_0, E_1 \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E_1' \mid \mathbf{E}], \mathbf{C}\rangle}$$

Before popping the current execution context E_0, the automaton spreads its θ_0 substitution to the parent execution context E_1—as θ_0 can contain more assignments than θ_1.

Finally, the following transition rule handles the case in which the current execution context contains a non-empty stream of goals $\bar{g}$. When this is the case the automaton applies the most recent substitution θ to all the goals in $\bar{g}$ before moving into Primitive Selection location and tries then to prove the truth of the first sub-goal in $\bar{g}$:

$$\frac{E = (\theta, \bar{g}, [], []) \qquad E' = (\theta, \bar{g}', [], []) \qquad \bar{g}' = \bar{g}/\theta}{\langle \mathsf{Goal\ Selection}, [E \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \mathsf{Primitive\ Selection}, [E' \mid \mathbf{E}], \mathbf{C}\rangle}$$

where by $\bar{g}/\theta$ we mean the application of substitution θ to all goals in $\bar{g}$.

Primitive Selection. The purpose of the Primitive Selection location is to select a primitive in the primitive store I in order to prove a (sub-)goal—provided a primitive matching the goal's signature exists in I. If this is the case, the primitive is triggered and the *first* primitive response is handled. Thus, there are three relevant situations handled in this location, by as many transition rules. A pivotal role in discriminating among situations is played by the first goal g of the current execution context. If primitive p is indexed in I via signature of g, then p is triggered and a response stream is generated in return. Only the first item in the stream, r, is consumed. This can be either an exception or a substitution. Each case is handled by a different transition rule. Otherwise, if the signature of g matches no primitive in I, no primitive is selected and, via subsequent transitions, a rule for the same goal is searched.

In particular, the following transition rule handles the case where a primitive p is found in I and the first response r is a substitution. When this is the case, a new execution context is pushed on the stack – for handling the first response –, and a new choice point is appended to queue—for handling any further response. After that, the automaton moves to the Primitive Execution location.

$$\frac{\begin{array}{c} E_0 = (\theta, \bar{g}, [], []) \quad \bar{g} = [g \mid \bar{g}'] \quad (f, n) = \mathit{signature}(g) \quad (f, n, p) \in I \quad [r \mid \bar{r}] = p(K, g) \\ r \in \Theta \quad E_1 = (\theta, \bar{g}, [], [r \mid \bar{r}']) \quad E_2 = (\theta, \bar{g}, [], \bar{r}') \quad \mathbf{C}' = \mathit{append}(\mathbf{C}, [E_2, E_0 \mid \mathbf{E}]) \end{array}}{\langle \mathsf{Primitive\ Selection}, [E_0 \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \mathsf{Primitive\ Execution}, [E_1, E_0 \mid \mathbf{E}], \mathbf{C}'\rangle}$$

Conversely, the following transition rule handles the case in which the first primitive response r is an exception. This implies that there could *not* be any further response in the response stream and the exception needs to be handled. For this reason, the automaton moves into the Exception location leaving the choice-point queue unaffected:

$$\frac{E_0 = (\theta, [g \mid \bar{g}'], [], []) \quad (f, n) = \mathit{signature}(g) \quad (f, n, p) \in I \\ [x \mid \bar{r}] = p(K, g) \quad x \in \mathcal{X} \quad E_1 = (\theta, [g \mid \bar{g}'], [], [x \mid \bar{r}])}{\langle \mathsf{Primitive\ Selection}, [E_0 \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \mathsf{Exception}, [E_1, E_0 \mid \mathbf{E}], \mathbf{C} \rangle}$$

In this case as well a new execution context is pushed on the stack – in order to make the exception visible in Exception –, whereas no new choice point is created.

Finally, the following transition rules handle the case in which a primitive is available in I for the goal g. When this is the case, the automaton simply moves into the Exception location:

$$\frac{E = (\theta, [g \mid \bar{g}'], [], []) \qquad (f, n) = \mathit{signature}(g) \qquad (f, n, p) \notin I}{\langle \mathsf{Primitive\ Selection}, [E \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \mathsf{Rule\ Selection}, [E \mid \mathbf{E}], \mathbf{C} \rangle}$$

Primitive Execution. The purpose of the Primitive Execution location is to *lazily* handle the response streams produced by primitives. To this regard, there are three relevant situations, handled by as many transition rules. In fact, while consuming a stream of solution responses, the automaton may either encounter an empty stream, or a stream whose first element is either an exception, or a substitution. The latter case is the most interesting one, as the substitution must be kept into account in the next computational steps. Conversely, in the other cases, the response stream is interrupted, even if with different outcomes: while the lack of responses simply provokes backtracking, exceptions need to be handled accordingly.

In particular, the following transition rule handles the case where the first response in the stream is a unifier θ'. When this is the case, θ' is merged with the current execution context substitution θ and execution proceeds in the Goal Selection location.

$$\frac{E = (\theta, [g \mid \bar{g}], [], [\theta', \ldots]) \qquad \theta' \in \Theta - \{\bot\} \qquad E' = (\theta \cup \theta', \bar{g}, [], [])}{\langle \mathsf{Primitive\ Execution}, [E \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \mathsf{Goal\ Selection}, [E' \mid \mathbf{E}], \mathbf{C} \rangle}$$

It is worth to highlight how this transition rule simply consumes a *single* response in the stream. Subsequent responses may be consumed after backtracking. Thus, this is where the lazy semantics of primitives is realised.

Conversely, the following transition rule handles both the case where the first response in the stream is a failure (i.e. $\bot$), and the case of any empty response stream. In all such cases, the automaton simply moves into the Backtracking location.

$$\frac{E = (\theta, \bar{g}, [], \bar{r}) \qquad \bar{r} = [\bot, \ldots] \vee \bar{r} = [] \qquad E' = (\theta, \bar{g}, [], [])}{\langle \textsf{Primitive Execution}, [E \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \textsf{Backtracking}, [E' \mid \mathbf{E}], \mathbf{C}\rangle}$$

Finally, the following transition rule handles the case where the first response in the stream is an exception. When this is the case, the automaton simply moves into the Exception location.

$$\frac{E = (\theta, \bar{g}, [], [x, \ldots]) \qquad x \in \mathcal{X} \qquad E' = (\theta, \bar{g}, [], [x])}{\langle \textsf{Primitive Execution}, [E \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \textsf{Exception}, [E' \mid \mathbf{E}], \mathbf{C}\rangle}$$

Rule Selection. The Rule Selection location is for clauses what the Primitive Selection location is for primitives. It aims at querying the KB and selecting the rules to be executed to prove a particular (sub-)goal true, provided that no primitive has been selected to the purpose. Accordingly, three transition rules are defined, each one handling a particular situation. The most common situation here is that a number of rules $\bar{r}$ are selected from K in order to solve some goal g. However, there is a small set of goals for which a particular treatment is reserved. These are: `!` (the "cut"), `true`, `fail`, and `false`. The first two are always considered successful, while the others are always considered failed.

More precisely, the following transition rule takes care of goals such as `!` and `true`. As they must always be evaluated successfully, this transition simply makes the automaton move into the Goal Selection location, after consuming the current goal g_0. However, in the particular case of $g_0 \equiv !$, the transition rule also provokes the *cut* of all choice points, up to the one relative to goal g_1 (included):

$$\frac{\begin{array}{c} E_0 = (\theta_0, [g_0 \mid \bar{g}_0], [], []) \qquad g_0 \in \{\texttt{true}, !\} \qquad E_1 = (\theta_1, [g_1, \ldots], \ldots) \\ E'_0 = (\theta_0, \bar{g}_0, [], []) \qquad \mathbf{C}' = cut(C, g_1) \end{array}}{\langle \textsf{Rule Selection}, [E_0, E_1 \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E'_0, E_1 \mid \mathbf{E}], \mathbf{C}'\rangle}$$

where $cut : \mathcal{C}^* \times \mathcal{A} \to \mathcal{C}^*$ is the function cutting off choice points, up to a given goal.

Conversely, the following transition rule takes care of goals such as `fail` and `false`. As they must always be evaluated successfully, this transition simply makes the automaton move into the Backtracking location, after consuming the current goal g. This transition rule also handles the case where K is queried for all rules whose head unifies with the current goal g, but no one is found.

$$\frac{E = (\theta, [g, \ldots], [], []) \qquad g \in \{\texttt{false}, \texttt{fail}\} \vee get(K, g) = []}{\langle \textsf{Rule Selection}, [E \mid \mathbf{E}], \mathbf{C}\rangle \xrightarrow{\tau} \langle \textsf{Backtracking}, [E \mid \mathbf{E}], \mathbf{C}\rangle}$$

Finally, the following transition rule handles the general case where K is queried for all rules $\bar{h}$ whose head unifies with the current goal g. Assuming that $\bar{h}$ contains at least one rule h, the automaton must then move to location Rule Execution, after pushing a new execution context on the stack – aimed at handling h –, and adding a new choice point to the queue—aimed at handling any further rule in $\bar{h}$:

$$\frac{\begin{array}{c} E_0 = (\theta, \bar{g}, [], []) \quad \bar{g} = [g \mid \bar{g}'] \quad g \in \mathcal{A} - \{\texttt{true}, \texttt{false}, \texttt{fail}, !\} \\ get(K, g) = \bar{h} \quad refresh(\bar{h}) = [h \mid \bar{h}'] \\ E_1 = (\theta, \bar{g}, [h \mid \bar{h}'], []) \quad E_2 = (\theta, \bar{g}, \bar{h}', []) \quad \mathbf{C}' = append(\mathbf{C}, [E_2, E_0 \mid \mathbf{E}]) \end{array}}{\langle \textsf{Rule Selection}, [E_0 \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \textsf{Rule Execution}, [E_1, E_0 \mid \mathbf{E}], \mathbf{C}' \rangle}$$

where $refresh : \mathcal{H}^* \to \mathcal{H}^*$ is a function refreshing all variables of all clauses in a clause stream/list.

Rule Execution. The Rule Execution location is for clauses what the Primitive Execution location is for primitives. Thus, the purpose of this location is to *lazily* handle the rule streams produced by KB. Accordingly, two transition rules are defined, each one handling a particular situation. In both situations, the current execution context is assumed to carry a non-empty rule stream to be handled. One situation concerns the case where the first rule in the stream has a head matching the current context goal. In this case, the execution can go on and focus on the body of that rule. The other situation concerns the opposite case, where execution must proceed with backtracking.

Accordingly, the following transition rule handles the first situation. The current execution context's first goal is g and the first rule is h. Provided that the head of h unifies with g, and letting θ' be their unifier, the current execution context is updated in such a way that the new substitution is $\theta \cup \theta'$ and the new goal stream contains all the atoms from the body of h, subject to the substitution θ'. After that, the automaton moves to the Goal Selection location.

$$\frac{\begin{array}{c} E = (\theta, [g \mid \bar{g}], [h, \ldots], []) \quad h = (a \leftarrow a_1, \ldots, a_n) \quad \theta' = mgu(g, a) \neq \bot \\ E' = (\theta \cup \theta', [g_1/\theta', \ldots, g_n/\theta'], [], []) \end{array}}{\langle \textsf{Rule Execution}, [E \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E' \mid \mathbf{E}], \mathbf{C} \rangle}$$

Conversely, the following transition rule handles the case where the head of h does not unify with g. In this case, the automaton simply moves into the Backtracking location.

$$\frac{E = (\theta, [g \mid \bar{g}], [h, \ldots], []) \quad h = (a \leftarrow \ldots) \quad mgu(g, a) = \bot \quad E' = (\theta, [g \mid \bar{g}], [], [])}{\langle \textsf{Rule Execution}, [E \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \textsf{Backtracking}, [E' \mid \mathbf{E}], \mathbf{C} \rangle}$$

Backtracking. The Backtracking location is the key point where the lazy consumption of rules and responses streams is performed by Prolog solvers. More precisely, this is where the choice points previously accumulated by the automaton in the choice-points queue are handled. Following this purpose, the Backtracking location may encounter three relevant situations, all depending on the content of the queue. The first situation concerns the case of the queue is empty, meaning that a new negative solution should be produced by the solver. The other situations concern the cases where the next choice point is carrying a non-empty rule or primitive response stream, respectively. In these cases, the automaton should move into either the Rule or Primitive Execution locations, in order to go on with resolution and consume the next rule or response.

Accordingly, the following rule handles the case where the choice-point queue is empty and the and the automaton should just move into the End final location, yield a new negative solution $\perp$.

$$\frac{}{\langle \mathsf{Backtracking}, \mathbf{E}, [] \rangle \xrightarrow{\perp} \langle \mathsf{End}, \mathbf{E}, [] \rangle}$$

Conversely, the following rule handles the case where the next choice point C in the queue is an execution context carrying a non-empty primitive response stream $\bar{r}$. When this is the case, the automaton simply adopts C as the next execution context, popping it from the choice-point queue and moving into the Primitive Execution location.

$$\frac{C = [(\theta, \bar{g}, [], \bar{r}), \ldots] \qquad \bar{r} \neq []}{\langle \mathsf{Backtracking}, \mathbf{E}, [C \mid \mathbf{C}] \rangle \xrightarrow{\tau} \langle \mathsf{Primitve\ Execution}, C, \mathbf{C} \rangle}$$

Finally, the following rule handles the case where the next choice point C in the queue is an execution context carrying a non-empty rule stream $\bar{h}$. When this is the case, the automaton simply adopts C as the next execution context, popping it from the choice-point queue and moving into the Rule Execution location.

$$\frac{C = [(\theta, \bar{g}, \bar{h}, []), \ldots] \qquad \bar{h} \neq []}{\langle \mathsf{Backtracking}, \mathbf{E}, [C \mid \mathbf{C}] \rangle \xrightarrow{\tau} \langle \mathsf{Rule\ Execution}, C, \mathbf{C} \rangle}$$

Exception Handling. The Exception location has the purpose of managing exceptions possibly raised by primitive responses in the Standard Prolog way—i.e. by climbing the proof tree towards the root, looking for a `catch/3` (sub-)goal whose second argument unifies with the raised exception and setting its third argument as the next sub-goal to be proved.

Following this purpose, the Exception location may encounter two notable situations. In both ones, the current execution is assumed to be carrying an exception

$x \in \mathcal{X}$. The first situation concerns the case where the exception can be *caught* since there exists on the stack an execution context of the form `catch/3` which may intercept the exception and let resolution continue. The second situation concerns the opposite case where the exception *cannot* be caught – since no such an execution context is contained into the stack – and resolution must be therefore interrupted.

In particular, the following transition rule handles the first situation where an execution context E' exists on the stack whose first goal is $\texttt{catch}(g_1, g_2, g_3)$. When this is the case, we denote by θ' the MGU among x and g_2. Then, the automaton pops from the stack all execution contexts up to E' (included), pushes a new execution context carrying g_3 as the first goal, and moves into the Goal Selection location.

$$\frac{\begin{array}{c} E = (\theta, [g \mid \bar{g}], [], [x, \ldots]) \quad x \in \mathcal{X} \quad E' = (\vartheta, [\texttt{catch}(g_1, g_2, g_3), \ldots], \bar{h}, \bar{r}) \\ \theta' = mgu(x, g_2) \neq \bot \quad \theta'' = \vartheta \cup \theta' \quad E'' = (\theta'', [g_3/\theta''], [], []) \end{array}}{\langle \textsf{Exception}, [E, \ldots, E' \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E'' \mid \mathbf{E}], \mathbf{C} \rangle}$$

Conversely, the following transitions rule handles the opposite situation where no execution context on the stack carries $\texttt{catch}(g_1, g_2, g_3)$ as the first goal—or, if it does, g_2 does not unify with x. When this is the case, the automaton simply moves into the Halt location, yielding an exceptional solution to the users.

$$\frac{\begin{array}{c} E = (\theta, [g \mid \bar{g}], [], [x, \ldots]) \qquad x \in \mathcal{X} \\ \mathbf{E} \neq [E, \ldots, (\vartheta, [\texttt{catch}(g_1, g_2, g_3), \ldots], \bar{h}, \bar{r}), \ldots] \end{array}}{\langle \textsf{Exception}, [E \mid \mathbf{E}], \mathbf{C} \rangle \xrightarrow{x} \langle \textsf{Halt}, [E \mid \mathbf{E}], \mathbf{C} \rangle}$$

Next Solutions. Both the End and Halt locations are final, meaning that the automaton reaches them immediately after the production on a novel solution—be it positive, negative, or exceptional. However, while the Halt location is a sink certainly provoking the automaton termination, the End state may allow the execution to be resumed. In particular, when the automaton is in location End, the users may trigger the automaton again looking for further solutions—which *may* be available if the choice-point queue is not empty.

Accordingly, the following transition rule handles the case where the automaton execution is resumed: once in location End the automaton may move back into location Backtracking, provided that the current choice-point queue $\mathbf{C}$ is non-empty.

$$\frac{\mathbf{C} \neq []}{\langle \textsf{End}, \mathbf{E}, \mathbf{C} \rangle \xrightarrow{\tau} \langle \textsf{Backtracking}, \mathbf{E}, \mathbf{C} \rangle}$$

Thus, the automaton actually terminates in location End only when the choice-

Listing 10.1: Interface of a general purpose `Solver` in 2P-Kt

```kotlin
interface Solver {
    val staticKb: Theory
    val dynamicKb: Theory
    val libraries: Libraries
    fun solve(goal: Struct): Sequence<Solution>
}
```

point queue is empty.

10.3 Predicates as Streams in 2P-Kt

In order to demonstrate the feasibility of our approach, we propose a case study based on 2P-Kt. 2P-Kt [CCO21a] is a Kotlin-based ecosystem for LP, including general API for stream-oriented logic solvers of any sort. Regardless of the particular logic, inference rule, or search strategy of choice, a logic solver is modelled in 2P-Kt as a prosumer of streams: it produces output streams of solutions and consumes input streams generated by primitives. A Prolog solver implementation is available as well, leveraging the state-machine-based design presented in section 10.2. Furthermore, 2P-Kt involves an API for writing primitives in Kotlin, by blending an imperative, object-oriented, and functional programming style .

In this section, we first illustrate briefly the portion of the 2P-Kt API involving solvers and primitives, then we discuss an example primitive implementing the TSP example from section 10.1.4.

10.3.1 2P-Kt Solvers and Primitives API

Here we focus on the resolution-related portion of the 2P-Kt API (cf. chapter 9 for further details). There, logic solvers are modelled as instances of the `Solver` type defined as shown in listing 10.1: Essentially, a logic solver is any entity exposing a method `solve` which accepts a logic `Structure` – i.e., a particular case of logic `Term` in the 2P-Kt type system – as the input goal, and produces a `Sequence` – i.e., a *lazy* stream in the Kotlin type system – of logic `Solutions` as output. Furthermore, 2P-Kt requires each logic solver to be composed by at least three more entities, namely: *(i)* a `staticKb` and *(ii)* a `dynamicKb`, both of type `Theory` – that is, an ordered and indexed container of logic clauses, retrievable via unification –, and *(iii)* a `libraries` container of type `Libraries`—which, within the scope of this section, is essentially an implementation of the structure indexing primitives.

Each `Solution` in 2P-Kt may be of any of three sorts, namely `Yes`, `No`, and

Listing 10.2: Interface of a general-purpose `Primitive` in 2P-Kt

```
fun interface Primitive {
    fun solve(request: Request<ExecutionContext>): Sequence<Response>
}
```

Listing 10.3: API of a primitive's `Request` in 2P-Kt

```
class Request(
    val context: ExecutionContext,
    val signature: Signature,
    val arguments: List<Term>
) {
    fun solve(subQuery: Struct): Sequence<Solution>
    fun replySuccess(): Response
    fun replyFail(): Response
    fun replyWith(substition: Substitution): Response
    fun replyException(exception: TuPrologRuntimeException): Response
}
```

`Halt`, representing the positive, negative, and exceptional case, respectively. All solutions carry the original query they are answering to, other than the `Substitution` they are answering through. So for instance, objects of type `Solution.Yes` always contain an object of type `Substitution.Unifier`, whereas other sorts of solutions always contain an object of type `Substitution.Fail`. Similarly, objects of type `Solution.Halt` carry the uncaught exception which interrupted the resolution process.

Primitives are modelled in 2P-Kt as functional interfaces, as shown in listing 10.2, i.e. as functions accepting a `Request` as input and returning a `Sequence` of `Responses` as output. There, `Request` (cf. listing 10.3) is a container of all the information needed at runtime to produce a sequence of `Responses`. These include: *(i)* a snapshot of `ExecutionContext` at invocation time – in turn including a snapshot of the solver's `staticKb` and `dynamicKb` –, *(ii)* the `Signature` of the invoked primitive, and *(iii)* the `List` of `Terms` storing actual `arguments` provided to the primitive upon invocation. Furthermore, each instance of `Request` exposes a bunch of methods – namely, the many `reply*()` ones –, aimed at generating a new `Response` for that particular `Request`. As `Responses` are mere containers of `Solutions`, there are many variants of the `reply*()` methods, each one aimed at generating a given sort of responses – e.g. responses carrying positive/negative/exceptional solutions – for the sub-goal that triggered the primitive. Finally, each request supports the spawning of an inner resolution process via its `solve(...)` method. This method creates a novel sub-solver through which primitive implementers can resolve sub-queries as part of some primitive execution.

Thanks to this design, any Kotlin function of the form shown in listing 10.4

Listing 10.4: General structure of a function acting as primitive in 2P-Kt

```
fun method(request: Request): Sequence<Response> = sequence {
  request.arguments[i] // read the i-th actual argument
  request.context.staticKb[h] // read clauses in KB whose head matches h
  solve(goal) // perform sub-queries

  val substitution = (arg0 mguWith value0) + (arg1 mguWith value1) + ...

  yield(request.replyWith(substitution))
  // or
  yield(request.replyFail())
  // or
  yield(request.reply*(...))
}
```

Listing 10.5: Implementation of a primitive aimed at lazily generating all natural numbers in 2P-Kt

```
fun natural(request: Request): Sequence<Response> = sequence {
    var n = 1
    while (true) {
        yield(Integer.of(n))
        n++
    }
  }.map {
    request.replyWith(request.arguments[0] mguWith it)
  }
```

can be considered a primitive in the eyes of a logic solver. This leverages a particular feature of Kotlin, namely the `sequence { ... }` blocks, which let developers write stream *primitives* by blending the imperative and functional programming styles. This is possible because of the `yield(value)` method which users may call inside `sequence { ... }` blocks in place of `return value` to provide values to the stream.

So, for instance, to implement the predicate `natural/1` – which holds true for all natural numbers –, one may write the primitive from listing 10.5. A Prolog solver would then treat such a primitive as a backtrackable predicate. Thus, in Prolog, one may use the goal `natural(X)` to enumerate all the natural numbers. Similarly, to implement the predicate `even/1` – which holds true for all *even* natural numbers –, one may simply rewrite method `natural` as follows:

Summarising, 2P-Kt primitives API supports the creation of backtrackable Prolog predicates out of lazy data streams.

Listing 10.6: Implementation of a primitive aimed at lazily generating all *even* natural numbers in 2P-Kt

```
fun even(request: Request): Sequence<Response> = sequence {
    var n = 1
    while (true) {
        yield(Integer.of(n))
        n++
    }
  }.filter {
    it.value % 2 == 0
  }.map {
    request.replyWith(request.arguments[0] mguWith it)
  }
```

10.3.2 Travelling Salesman Problem in 2P-Kt

The real potential of primitives is revealed when they are exploited by solvers to manage input data streams from the external world. There, the external world may be any source of data, there including other solvers, possibly of different nature. For example, primitives may be exploited to let a Prolog solver call a TSP solver to efficiently compute solutions for TSP instances, as discussed in section 10.1.4. Accordingly, here we demonstrate how a primitive of such a sort may be realised through 2P-Kt.

In [Cia21] we provide a GitHub repository hosting the source code of a 2P-Kt primitive leveraging Google OR-Tools [PF19] to efficiently solve TSP instances. Google OR-Tools is a C++ library proving many constraint programming and operative research tools – there including routing-related facilities –, and some JVM bindings which let us exploit such tools in Kotlin.

Accordingly, our repository includes some scripts aimed at automating the compilation and execution of a simple demo involving a command-line TSP-enabled Prolog interpreter. Following the discussion from section 10.1.4, such a Prolog interpreter exposes a `tsp/3` predicate aimed at enumerating the minimally-costly circuits for any given set of cities, provided that the interpreter's KB contains several `path/3` facts describing the connections among those cities. As an ordinary Prolog interpreter, such facts may be either consulted from a `.pl` file or dynamically asserted via `assert/1`.

The actual operational behaviour of predicate `tsp/3` is governed by the `Tsp` primitive whose source code (stub) is shown in listing 10.7 (cf. [Cia21] for full source code). The `Tsp` primitive is a singleton object of type `TernaryRelation` – i.e., a particular sort of `Primitive`, tailored on ternary predicates –, whose main behaviour is encapsulated within the `computeAll` method.

The `Tsp` object is also endowed with a method – namely, `tsp` – which returns a sequence of circuits and costs for any given list of cities provided as input. Such

Listing 10.7: 2P-Kt primitive implementing the `tsp/3` predicate

```kotlin
import it.unibo.tuprolog.core.List as LogicList

object Tsp : TernaryRelation<ExecutionContext>("tsp") {
  init { com.google.ortools.Loader.loadNativeLibraries() }

  private fun Request<ExecutionContext>.tsp(cities: List<Term>): Sequence<Pair<LogicList, Integer>> { ... }

  // other utility methods

  override fun Request<ExecutionContext>.computeAll(fst: Term, snd: Term, trd: Term): Sequence<Response> {
    val allCities = solve(Struct.template("path", 3))
      .filterIsInstance<Solution.Yes>()
      .map { it.solvedQuery }
      .flatMap { sequenceOf(it[0], it[1]) }
      .toSet()

    return allCities
      .subsets()
      .flatMap { it.permutations() }
      .map { it to (Set.of(it) mguWith fst) }
      .filter { (cities, substitution) -> cities.isNotEmpty() && substitution is Unifier }
      .flatMap { (cities, substitution) -> tsp(cities).map { it.addLeft(substitution) } }
      .map { (substitution, circuit, cost) -> substitution + (snd mguWith circuit) + (trd mguWith cost) }
      .filterIsInstance<Unifier>()
      .map { replySuccess(it) }
  }
}
```

method assumes each input city to be represented by a logic term – in particular, a constant –, and outputs circuits represented as logic lists of cities represented in the same way. Behind the scenes, the `tsp` interacts both the Prolog interpreter's KB to read distances among cities, and a Google OR-Tool solver for computing all possible solution to a particular TSP instance.

The `computeAll` handles the situation where the Prolog interpreter meets a (sub-)goal of the form `tsp(Cities, Circuit, Cost)`—where all variables may be partially or totally uninstantiated. The method operation can then be described as a pipeline of *lazy* operations applied to the actual arguments of `tsp/3`, which we refer as `fst`, `snd`, and `trd` within the method. Accordingly, the method firstly performs a sub-query aimed at computing the set of all cities currently contained into the KB (cf. variable `allCities` in listing 10.7). The sub-query is a Prolog goal of the form `path(_, _, _)`, whose solutions are all eagerly consumed and their first and second arguments – which are assumed to be city names – are merged into a set, to remove duplicates. Then, all possible permutations of all possible subsets of `allCities` are lazily generated. However, only the subsets of cities that unify with `fst` are actually selected (this may be just one set of cities if `fst` refers to a fully instantiated set of cities) for the next steps of the computation. Then, for all selected sets of cities, all possible solutions to the corresponding TSP instance are computed. Finally, each possible circuit (resp. cost) computed for each TSP instance is unified with `snd` (resp. `trd`). Failed unifications are of course dropped, while the successful ones are converted into responses of the `tsp/3` primitive.

It is worth to highlight that the whole pipeline is *lazy*. This implies that even once the first TSP solution has been presented to the user, the other ones are still to be computed.

10.4 Recap and Research Perspectives

In this chapter we address the issue of stream processing in logic programming.

In particular, we discuss how logic solvers can be naturally conceived as lazy prosumers of data streams as they *(i)* lazily *produce* data streams thanks to their interactive nature, *(ii)* lazily *consume* data streams as part of their resolution process—e.g. when they access knowledge bases. Furthermore, we show how logic solvers can support the processing of input data stream via the notion of predicates as *primitives*, which we introduce in this chapter. Summarising, primitives are reactive computational units which logic solvers may trigger so as to receive data streams from the external world. This may be useful, for instance, to let a solver delegate some part of its resolution process to some external entity—assuming that it is optimised to the purpose.

To demonstrate the feasibility of our approach in the specific (and technically most relevant) case of Prolog, we propose a primitive-enabled modelling of Prolog solvers as state machines, formalising the lazy consumption of streams via backtracking. The proposed formalisation preserves the standard operation of Prolog and requires no modification to the language, while enabling Prolog solvers to process data streams.

Finally, we discuss the use case of 2P-Kt [CCO21a], a Kotlin-based technology for LP including an implementation of Prolog solvers relying on our state-machine-based formalisation. We then exploit 2P-Kt to show how primitives can be used to bridge different sorts of solvers together via a few lines of Kotlin code.

In our perspective, this work represents one further step towards the *practical* exploitation of LP – and, in particular, Prolog – as a general means for stream processing. Notably, our contribution presents some similarities with other works [TWS19, Red16]. In particular, similarly to [TWS19], we focus on letting Prolog manipulate streams of data; while, similarly to [Red16], we provide a mechanism to let logic solvers delegate computations to external entities. However, differently from [TWS19], we require no variation to the syntax, functioning, or libraries of Prolog; while, unlike [Red16], we focus on Prolog rather than ASP.

A number of issues remain uncovered in this work, and will be the subject of our future research. Among the many, the most relevant issues concern *time*, *side effects*, and *concurrency*. In particular we plan to explore the temporal dimension in LP-based stream processing, by providing for instance some means to support time-dependent or time-limited data streams. Similarly, we would like to explore the intricacies related to the processing of data streams which may affect the internal state of a logic solver – e.g. by affecting the KB – in a predictable way. Finally, as further discussed in section 15.1, we are interested developing a framework for *concurrent* resolution, following the same state-machine-based approach we exploit in this chapter.

Chapter 11

Bridging LP and Mainstream Programming Paradigms

This chapter contains contributions from the following works of ours: [CCS+20]

Logic Programming (LP) [Apt01, Kow74] is a programming paradigm based on formal logic, inspired to the idea of declaratively specifying a program semantics via logic formulæ, so that automatic reasoners can then prove such formulæ by leveraging on different *control* strategies. In the years, LP has contributed to the development of a number of diverse research fields laying under the umbrella of symbolic AI—such as automatic theorem proving, multi-agent systems, model checking, research optimisation, natural language processing, etc.

Nowadays, LP is one of the major programming paradigms available for software development, along with the imperative, functional, and object-oriented ones. In particular, LP is today one of the best-suited choices for tackling problems involving knowledge representation, logic inference, automated reasoning, search in a discrete space, or meta-programming [CCDO20]. Moreover, today LP supports the core of AI components in pervasive and distributed systems, providing intelligence where and when it is needed [OC19].

This is why the *integration* of LP within the main programming languages is nowadays more interesting than ever. However, to make it actually work, integration should be designed – from a linguistic standpoint – so as to reduce both the development time and the learning curve of developers, as well as the psychological and cultural resistances against the adoption of new paradigms—thus, basically, moving LP up to a manageable level by the OOP developer.

Most mainstream programming languages – such as Java, Kotlin, Scala, Python, JavaScript, C# – have recognised the added value of the integration of diverse programming paradigms under a unique syntax, a coherent API, and a rich standard library. In fact, they all already support both the object-oriented (OOP) and functional (FP) programming paradigms. We believe that the same languages would largely benefit from extending their support to LP languages as well, making them usable in the OOP context in the same way as FP features already are.

Interoperability among Prolog [CR93] (as the first and most prominent LP language) with other languages from different paradigms is not new: the Prolog community has actually been studying this theme for years [BC02], as shown by the many forms of integration historically present in most Prolog systems, providing either *(i)* logic-to-OOP interoperability, making object orientation available within Prolog scripts, or *(ii)* OOP-to-logic interoperability, making logic programming exploitable within object-oriented code—or both, as in the case of `tu`Prolog Java Library [DOR05]. It is worth noting here that existing integrations make quite strong assumptions. For instance, item *(i)* assumes the main application is written in Prolog and the interoperation is given via a suitable interface allowing the injection of (small) parts written in other languages, while item *(ii)* implicitly assumes LP and Prolog to be somehow *harmonised* with the language(s) hosting them, at the paradigm, syntactical, and technological levels. Both these assumptions can actually create a barrier against an effective adoption of LP in today applications despite its huge potential.

Therefore, the approach adopted in this work is to devise instead a form of LP-FP-OOP *blended integration*, such that the key features of LP paradigm can be exposed and made available to mainstream language developers in a way that is the most natural in that context. The key requirement, therefore, is the "making LP easy to adopt for OOP programmers", in order to break down the learning curve for non-experts, and pursuit the typical developers' mindset. Going beyond the mere interoperability intended as simple technical habilitation, our approach is meant to embrace the perspective of the OOP developer aiming at exploiting the potential of LP.

To this end, we show how designing Prolog as a *domain-specific language* (DSL) for an OOP language such as Kotlin can make LP close enough to the OOP developers' mindset to overcome most cultural barriers, while at the same mimicking the Prolog syntax and semantics close enough to make its use straightforward for LP developers.

Generally speaking, the integration of Prolog with other paradigms aims at bringing the effectiveness and declarativeness of LP into general-purpose OOP framework. In particular, OOP designers and programmers can be expected to benefit from LP features for *(i)* data-driven or data-intensive computations, such as in the case of complex-event-processing frameworks; *(ii)* operation research or symbolic AI algorithms, or more generally other algorithms involving a search space to be explored; *(iii)* multi-agent systems, and the many areas in that field leveraging on LP, such as knowledge representation, argumentation, normative systems, etc.

The proposed solution is based on the 2P-Kt technology [2P-21, CCO21a], a re-engineering of the tuProlog project [tuP21, DOR01] as a Kotlin multi-platform library supporting the JVM, JS, Android, and Native platforms. The case of a Kotlin-based Prolog DSL is presented and discussed.

Accordingly, the chapter is organised as follows. Section 11.1 briefly introduces LP, providing details about tuProlog, 2P-Kt, and Kotlin as well. It also summarises the current state of the integration between LP and mainstream programming languages. Section 11.2 discusses the rationale, design, and architecture of our Prolog-Kotlin DSL along with some examples. Section 11.3, briefly illustrates a case study where our DSL is used in combination with functional programming to solve a simple AI task in an elegant way. Finally, in section 11.4, concludes the chapter by providing some insights about the possible future research directions stemming from our work.

11.1 Background

11.1.1 LP integration with other languages

In order to integrate LP with high-level languages, many Prolog implementations expose a foreign language interface (FLI, table 11.1) towards some high-level *target* language. In most cases, the target language is Java, and the FLI is bi-directional—meaning that it supports both "calling Prolog from the target language" and vice versa. In a few particular cases, however, the JavaScript and C# languages are also supported. As we are mostly interested in discussing how and to what extent Prolog exploitation is possible within the target languages, in the reminder of this section we only focus on those FLI allowing to "call Prolog from the target language".

Each FLI mentioned in table 11.1 is characterised by a number of features that heavily impact the way the users of the target languages may actually exploit Prolog. These are, for instance, the nature of the FLI – which is tightly related to the target platform of the underlying Prolog implementation – and its reference programming paradigm.

By "nature" of the FLI, we mean the technological mechanism exploited by a particular Prolog implementation to interact with the target language. There are some cases – like tuProlog and τProlog – where this aspect is trivial, because the target language is also the implementation language—so Prolog is considered there as just another library for the target language. More commonly, however, Prolog is implemented in C, and the Java Native Interface (JNI) is exploited to make it callable from Java. This is for instance the case of SWI- and ECLiPSe-Prolog. While this solution is very efficient, it hinders the portability of Prolog into par-

Table 11.1: Prolog implementations and their foreign language interfaces (FLI)

Prolog Implementation	Platform	FLI towards	Nature of FLI	Paradigm of FLI	Source
BProlog [BPr21]	C	Java	JNI	imperative	Official BProlog doc.
Ciao! [Pro21a]	C	Java	TCP/IP	imperative	Official Ciao Prolog doc.
ECLiPSe [Pro21b]	C	Java	JNI	imperative	Official ECLiPSe doc.
SICStus [SP21]	C	Java, C#	TCP/IP	object-oriented	Jasper Library
SWI [Pro21c]	C	Java	JNI	object-oriented	JPL API
τProlog [Pro21d]	JS	JavaScript	Native	object-oriented	Project homepage
`tu`Prolog [tuP21]	JS, JVM Android	Kotlin, Java, JavaScript	Native	object-oriented, functional	Project homepage
XSB [Pro21e]	C	Java	TCP/IP	imperative	Interprolog Java Bridge

ticular contexts such as Android, and its exploitation within mobile applications.

Another viable solution is to leverage on the TCP/IP protocol stack. This is for instance the case of SICStus- and Ciao-Prolog. The general idea behind this approach is that the Prolog implementation acts as a remote server offering logic-programming services to the target language via TCP/IP, provided that a client library exists on the Java side making the exploitation of TCP/IP transparent to the users. While this solution is more portable – as virtually any sort of device supports TCP/IP –, it raises efficiency issues because of the overhead due to network/inter-process communications. A more general solution of that sort is instead offered by the LPaaS architecture [CDMO18], where the fundamental idea is to have many Prolog engines distributed over the network, accessed as logic-based web services. By the way, the implementation of LPaaS[1] is based on `tu`Prolog.

By "reference programming paradigm" of the FLI we mean the specific programming style proposed by a Prolog implementation to the target language users through its API. Some FLI are conceived to be used in a strictly imperative way: this is e.g. the case of BProlog or Ciao! Prolog, where a Java API is available for writing Prolog queries and issue them towards the underlying Prolog system in an imperative way. Other Prolog implementations, such as SICStus- and SWI-Prolog, offer a more object-oriented API which let developer not only represent queries but also terms, and solutions (there including variable bindings) which can be consumed through ordinary OOP mechanisms such as iterators.

In most cases, however, the code to be produced in the target language is far less concise and compact than pure Prolog – unless strings and parsing are extensively adopted –, especially when the size of the Prolog code to be represented grows. So, for instance, the simple Prolog query `?- parent(adam, X)` (aimed at computing who Adam's son is) in Java through SWI-Prolog's JPL interface would be written as in listing 11.1. While this a totally effective solution on the technical level,

[1]`http://lpaas.apice.unibo.it`

Listing 11.1: SWI-Prolog JPL interface example

```java
Query query = new Query("parent",
                new Term[] {
                    new Atom("adam"),
                    new Variable("X") } );
Map<String,Term> solution = query.oneSolution();
System.out.println("The child of Adam is " + solution.get("X"));
```

we argue that a more convenient integration among LP and the other paradigms is possible. In particular, in this chapter we show how 2P-Kt allows for a finer integration at the paradigm level through *domain-specific languages*.

11.1.2 Kotlin Domain-Specific Languages (DSL)

Domain-specific languages (DSL) are a common way to tackle recurrent problems in a general way via software engineering. When a software artefact (e.g. library, architecture, system) is available to tackle with some sort of problems, designers may provide a DSL for that artefact to ease its usage for practitioners. There, the exploitation of a DSL hides the complexity of the library behind the scenes, while supporting the usage of the artefact for non-expert users too. This is, for instance, the approach of the Gradle build system[2]—which is nowadays one of the most successful build automation systems for the JVM, Android, and C/C++ platforms. It is also the approach followed by the JADE agent programming framework, which is nowadays usable through the Jadescript DSL [BCMP20].

Building a DSL usually requires the definition of a concrete syntax, the design and implementation of a compiler or code generator, and the creation of an ecosystem of tools—e.g., syntax highlighters, syntax checkers, debuggers. In particular, compilation or code generation are fundamental as they are what makes a DSL machine-interpretable and -executable.

Some high-level languages, however, such as Kotlin, Groovy, or Scala, enable a different approach. They come with a flexible syntax which *natively* supports the definition of new DSL with no addition to the hosting language required. For instance, Gradle consists of a JVM library of methods for building, testing, and deploying sources codes, plus a Kotlin- or Groovy-based DSL allowing developers to customise their particular workflows.

The advantages of this approach are manifold. First, no compiler or code generator has to be built, as the DSL is already part of the hosting language and it is therefore machine-interpretable and -executable by construction. Second, the DSL automatically inherits all the constructs, API, and libraries of the hosting

[2]`https://gradle.org`

language—there including conditional or iterative constructs, string manipulation API, etc., which are common and useful for many DSL, regardless of their particular domain. Third, the ecosystem of tools supporting the hosting language – e.g. compilers, debuggers, formatters, code analysers, IDE, etc. – can be reused for the DSL as well, easing its adoption and making it more valuable.

When Kotlin is the hosting language of choice, DSL leverage on a small set of features making the Kotlin syntax very flexible, described below:

operator overloading[3] — allowing ordinary arithmetic, comparison, access, and function invocation operators to change their ordinary meaning on a per-type basis;

block-like lambda expressions[4] — including a number of syntactic sugar options such as *(i)* the possibility to omit formal parameters in case of a single-argument lambda expression, and *(ii)* the possibility to omit the round parentheses in case of a function invocation having a lambda expression as a last argument;

function types/literals with receiver[4] — allowing functions and methods to accept lambda expressions within which the `this` variable references a different object than the outer scope;

extension methods[5] — allowing pre-existing types to be extended with new instance methods whose visibility is scope-sensible.

Of course, Kotlin-based DSL automatically inherit the full gamma of facilities exposed by the Kotlin language and standard library—there including support for imperative, and object-oriented programming, as well as a rich API supporting functional programming through most common high-order operations. Furthermore, if properly engineered, these DSL may be executed on all the platforms supported by Kotlin—which commonly include, at least, the JVM, JavaScript, and Android. This is for instance the case of our DSL proposed in section 11.2.

While this chapter focuses on Kotlin-based DSL, similar results could be achieved in other languages by means of equivalent mechanisms. For instance, in Scala, extension methods have to be emulated via implicit classes, and DSL, in particular, can be built via the "Pimp My Library" pattern [OMO10].

[3]`https://kotlinlang.org/docs/reference/operator-overloading.html`

[4]`https://kotlinlang.org/docs/reference/lambdas.html`

[5]`https://kotlinlang.org/docs/reference/extensions.html`

Listing 11.2: Prolog theory describing Abraham's family tree

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

parent(abraham, isaac).
parent(isaac, jacob).
parent(jacob, joseph).
```

11.2 A domain-specific language for LP

11.2.1 Design Rationale

Regardless of the technological choices, the design of our DSL leverages on a small set of principles ($\mathbf{P_i}$) briefly discussed below. In fact, our aim is to ($\mathbf{P_1}$) provide a DSL that is a *strict* extension of its hosting language, meaning that no feature of the latter language is prevented by the usage of our DSL. Dually, we require ($\mathbf{P_2}$) our DSL to be fully interoperable and finely integrated with the hosting language, meaning that all the features of the latter language can be exploited from within our DSL. However, we also require ($\mathbf{P_3}$) the DSL to be well encapsulated and clearly identifiable within the hosting language, in order to prevent unintended usage of the DSL itself. Finally, we require ($\mathbf{P_4}$) our DSL to be as close as possible to Prolog, both at the syntactic and semantic level, to ease its exploitation for logic programmers.

In order to accomplish to the aforementioned principles, we choose the Kotlin language as the technological reference for prototyping our proposal. This is because it *(i)* comes with a flexible syntax supporting the definition of DSL, *(ii)* supports a considerable number of platforms (JVM, JavaScript, Android, Native) and therefore enables a wide exploitation of LP – for instance, on smart devices –, *(iii)* includes a number of libraries supporting LP-based applications, such as 2P-Kt.

11.2.2 The Kotlin DSL for Prolog

Before dwelling into the details of our proposal, we provide an overview of what a Kotlin DSL for Prolog has to offer.

Let us consider the Prolog theory shown in listing 11.2 describing a portion of Abraham's family tree. It enables a number of queries, e.g. `ancestor(abraham, X)`, which can be read as "Does there exist some `X` which is a descendant of Abraham?". According to the Prolog semantics, this query may have a number of solutions, enumerating all the possible descendants of Abraham that can be deduced from the above theory—i.e., Isaac, Jacob, and Joseph.

Listing 11.3: Kotlin code generating the logic theory from listing 11.2

```kotlin
prolog {
  staticKb(
    rule {
      "ancestor"("X", "Y") `if` "parent"("X", "Y")
    },
    rule {
      "ancestor"("X", "Y") `if` ("parent"("X", "Z") and "ancestor"("Z", "Y"))
    },
    fact { "parent"("abraham", "isaac") },
    fact { "parent"("isaac", "jacob") },
    fact { "parent"("jacob", "joseph") }
  )

  for (sol in solve("ancestor"("abraham", "X")))
    if (sol is Solution.Yes)
      println(sol.substitution["X"])
}
```

The same result may be attained through the Kotlin program depicted in listing 11.3, which leverages on our DSL for Prolog: The program creates a Prolog solver and initialises it with standard built-in predicates. Then it loads a number of facts and rules representing the aforementioned Prolog theory about Abraham's family tree. Finally, it exploits the solver to find all the descendants of Abraham, by issuing the query `ancestor(abraham, X)`.

It is worth noting how the simple code snippet exemplified above is adherent w.r.t. our principles. In fact, the whole DSL is *encapsulated* ($\mathbf{P_3}$) within the

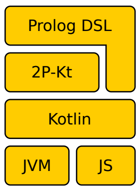
$$\texttt{prolog } \{\ \langle DSL\ \ block\rangle\ \}$$

In there, LP facilities can be exploited in combination with the imperative and functional constructs offered by the Kotlin language and its standard-library ($\mathbf{P_1}$ and $\mathbf{P_2}$)—such as the `for-each`-loop used to print solutions in the snippet above. Furthermore, within `prolog` blocks, both theories and queries are expressed via a Kotlin-compliant syntax mimicking Prolog ($\mathbf{P_4}$). The main idea behind such syntax is that each expression in the form

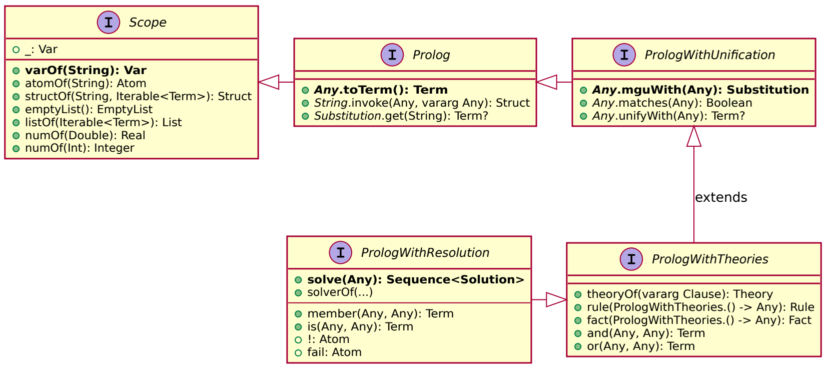
$$\texttt{"functor"}(\langle e_1\rangle,\ \langle e_2\rangle,\ \ldots)$$

is interpreted as a compound term (a.k.a. structure) in the form `functor`(t_1, t_2, ...), provided that each Kotlin expression e_i can be recursively evaluated as the term t_i—e.g. capital strings such as `"X"` are interpreted as variables, whereas non-capital strings such as `"atom"` (as well as Kotlin numbers) are interpreted as Prolog constants. In a similar way, expressions of the form[6]

[6]backticks make Kotlin parse words as identifiers instead of keywords



(a) Layered view showing the modules of 2P-Kt and our Kotlin DSL for Prolog


Scope
_: Var
varOf(String): Var
atomOf(String): Atom
structOf(String, Iterable<Term>): Struct
emptyList(): EmptyList
listOf(Iterable<Term>): List
numOf(Double): Real
numOf(Int): Integer
Prolog
Any.toTerm(): Term
String.invoke(Any, vararg Any): Struct
Substitution.get(String): Term?
PrologWithUnification
Any.mguWith(Any): Substitution
Any.matches(Any): Boolean
Any.unifyWith(Any): Term?
extends
PrologWithResolution
solve(Any): Sequence<Solution>
solverOf(...)
member(Any, Any): Term
is(Any, Any): Term
!: Atom
fail: Atom
PrologWithTheories
theoryOf(vararg Clause): Theory
rule(PrologWithTheories.() -> Any): Rule
fact(PrologWithTheories.() -> Any): Fact
and(Any, Any): Term
or(Any, Any): Term

(b) Structural view showing the API of our Kotlin DSL for Prolog, leveraging on the 2P-Kt API

Figure 11.1: Architectural view of our Kotlin DSL for Prolog

$$\texttt{rule \{ "head"(}\langle e_1\rangle\texttt{, ..., }\langle e_N\rangle\texttt{) `if` (}\langle e_{N+1}\rangle\texttt{ and ... and }\langle e_M\rangle\texttt{) \}}$$

are interpreted as Prolog rules of the form

$$\texttt{head}(t_1,\ \dots,\ t_N)\ \texttt{:-}\ t_{N+1},\ \dots,\ t_M$$

provided that $M > N$ and each Kotlin expression e_i can be recursively evaluated as the term t_i. A similar statement holds for fact expressions of the form `fact { ... }`.

To support our DSL for Prolog, a number of Kotlin classes and interfaces have been designed on top of the 2P-Kt library, exploiting manifold extensions methods, overloaded operators, etc. to provide the syntax described so far. The details of our solution – there including its architecture, design, and implementation – are discussed in the reminder of this section.

11.2.3 Architecture, Design, Implementation

The Kotlin DSL for Prolog is essentially a compact means to instantiate and use objects from the 2P-Kt library, through a Prolog-like syntax. More precisely, it consists of a small software layer built on top of Kotlin and 2P-Kt, as represented by fig. 11.1a. In particular, the DSL is enabled by the five *factory interfaces* depicted in fig. 11.1b. We call *factory interface* a type definition whose methods are aimed at *instantiating* objects of related types, as dictated by the Gang of Four' abstract factory pattern [GHJV95].

The five factory interfaces are: `Scope`, `Prolog`, `PrologWithUnification`, `PrologWithTheories`, and `PrologWithResolution`. Each factory type extends the previous one with more LP-related functionalities. So, for instance, while instances of `Scope` simply provide the basic bricks to create logic terms, instances of `Prolog` leverage on these bricks to enable the exploitation of the Prolog-like syntax exemplified above. `PrologWithUnification` extends `Prolog` with unification-related facilities, and it is in turn extended by `PrologWithTheories`, which lets developers create both clauses and theories via a Prolog-like syntax. Finally `PrologWithResolution` extends `PrologWithTheories` by adding resolution-related facilities, plus some syntactic shortcuts for writing rules exploiting Prolog standard predicates such as `member/2`, `length/1`, etc.

In the following we provide further details about how each factory contributes to our DSL.

Scope. The simplest factory type is `Scope`. As suggested by its name, it is aimed at building terms which must share one or more variables. For this reason, it exposes a number of factory methods – roughly, one for each sub-type of `Term` –, some of which are mentioned in fig. 11.1b. The main purpose of a scope, however, is to enable the creation of objects reusing the same logic `Variable`s more than once. Thus, it includes a method – namely, `varOf(String)` – which always returns the same variable if the same name is provided as input.

For example, to create the term `member(X, [X | T])`, one may write:

```
val m = Scope.empty {
    structOf("member", varOf("X"), consOf(varOf("X"), varOf("T")))
} // m references the term member(X, [X | T])
```

There, the two calls to `varOf("X")` actually return the same object, if occurring within the same scope—whereas they would return different variables if invoked on different scopes. This mechanism is what enables developers to instantiate terms and clauses without having to explicitly refresh variables among different clauses: it is sufficient to create them within different scopes.

Prolog. The `Prolog` factory type extends `Scope` by adding the capability of creating terms through a Prolog-like syntax. To do so, it exposes a number of extension methods aimed at automatically converting Kotlin objects into Prolog terms or using Kotlin objects in place of Prolog terms. The most relevant extension methods are mentioned in fig. 11.1b. These methods are:

- `fun Any.toTerm(): Term`, which is an extension method aimed at making any Kotlin object convertible into a Prolog term, through the syntax `obj.toTerm()`. To convert an object into a term, it leverages on the following type mapping: *(i)* a Kotlin number is either converted into a Prolog real or integer number, depending on whether the input number is floating-point or not, *(ii)* a Kotlin string is either converted into a Prolog variable or atom, depending on whether the input string starts with a capital letter or not, *(iii)* a Kotlin boolean is always converted into a Prolog atom, *(iv)* a Kotlin iterable (be it an array, a list, or any other collection) is always converted into a Prolog list, provided that each item can be recursively converted into a term, *(v)* a Kotlin object remains unaffected if it is already an instance of `Term`, *(vi)* an error is raised if the input object cannot be converted into a `Term`.

- `operator fun String.invoke(vararg Any): Struct`, which is an extension method aimed at overloading the function invocation operator for strings. Its purpose is to enable the construction of compound terms through the syntax `"f"(arg`$_1$`, ..., arg`$_N$`)`, which mimics Prolog. Its semantics is straightforward: assuming that each `arg`$_i$ can be converted into a term via the `Any.toTerm()` extension method above, this method creates a N-ary `Structure` whose functor is `"f"` and whose i^{th} is `arg`$_i$`.toTerm()`. So, for instance, the expression

  ```
  "member"("X", arrayOf(1, true))
  ```

  creates the Prolog term `member(X, [1, true])`.

- `fun Substitution.get(String): Term?`, which is an extension method aimed at overloading the `get` method of the `Substitution` type in such a way that is can also accept a string other than a `Variable`. This enables DSL users to write expressions such as

  ```
  substitution.get("X")
  ```

  instead of having to create a variable explicitly via `varOf("X")`. While we only discuss this method, the main idea here is that every method in the 2P-Kt API accepting some basic Prolog type – such as `Var`, `Atom`, or `Real` – as argument should be similarly overloaded to accept the corresponding Kotlin type as well—e.g. `String` or `Double`. This is what enables a fine-grained integration of our DSL with the Kotlin language and the 2P-Kt library.

It is worth highlighting that every `Prolog` object is also a particular sort of `Scope`. So, converting the same string into a `Variable` twice or more times, within the same `Prolog` object, always yields the exact same variable.

Prolog with unification. The `PrologWithUnification` factory type extends `Prolog` by adding the capability of *(i)* computing the most general unifier (MGU) among two terms, *(ii)* checking whether two terms match or not according to logic unification – i.e., checking if a MGU exists unifying the two terms –, and *(iii)* computing the term attained by unifying two terms—assuming an MGU exists for them. To do so, it exposes a number of extension methods aimed at providing unification-related support to Kotlin objects, provided that they can be converted into terms. The most relevant extension methods are mentioned in fig. 11.1b.

Prolog with theories. The `PrologWithTheories` factory type extends `PrologWithUnification` by adding the capability of creating logic clauses (e.g. rules and facts) and theories. To do so, it exposes a number of methods aimed supporting the conversion of Kotlin objects into Prolog clauses – through the syntactic facilities presented so far –, and their combination into theories. The most relevant methods, mentioned in fig. 11.1b, are the following:

- `fun theoryOf(vararg Clause): Theory`, an ordinary method aimed at creating a logic `Theory` out of a variable amount of clauses.
- `` infix fun Any.`if`(Any): Rule ``, which is an extension method aimed at creating logic rules via a Prolog-like syntax in the form `` head `if` body ``. Its semantics is straightforward: assuming that both `head` and `body` can be converted into logic goals via the `Any.toTerm()` extension method above, this method creates a binary `Structure` whose functor is `':-'` and whose arguments are `head.toTerm()` and `body.toTerm()`. Similar methods exists – namely, `and`, `or`, etc. – to create conjunctions or disjunctions of clauses.
- `fun rule(PrologWithTheories.()-> Any): Rule`, an ordinary method aimed at creating a rule in a separate scope, thus avoiding the risk of accidentally referencing the variables created elsewhere. It creates a fresh, empty, and nested instance of `PrologWithTheories` and accepts a function with receiver to be invoked on that nested instance. The function is expected to return a Kotlin object which can be converted into a Prolog rule. Any variable possibly created within the nested scope is guaranteed to be different than any homonymous variable defined elsewhere.
- `fun fact(PrologWithTheories.()-> Any): Fact`, analogous to the previous method, except that it is aimed at creating Prolog facts. So, for instance, one may write

```
val r1 = fact {
    "member"("X", consOf("X", `_`))
}
val r2 = rule {
    "member"("X", consOf(`_`, "T")) `if` "member"("X", "T")
}
```

while being sure that the X variable used in r1 is different than the one used in r2.

Prolog with resolution. The `PrologWithResolution` factory type extends `PrologWithUnification` by adding the capability of performing logic queries and consuming their solutions attained through the standard Prolog semantics. To do so, it exposes a number of methods aimed at supporting *(i)* the instantiation of Prolog `Solvers`, *(ii)* the loading of Prolog theories, either as static or dynamic knowledge bases (KB), and *(iii)* the invocation of Prolog queries on those KB. The most relevant methods, mentioned in fig. 11.1b, are the following:

- `fun solve(Any, Long): Sequence<Solution>`, which is an ordinary method aimed at executing Prolog queries without requiring a new solver to be explicitly created. It accept a Kotlin object as argument – which must be convertible into a Prolog query –, and an optional timeout limiting the total amount of time the solver may exploit to compute a solution. If the provided arguments are well formed, this method returns a `Sequence` of `Solutions` which *lazily* enumerates all the possible answers to the query provided as input, using Prolog's SLDNF proof procedure.

- `fun staticKb(vararg Clause)`, which is an ordinary method aimed at loading the *static* KB the `solve` method above will operate upon.

- `fun dynamicKb(vararg Clause)`, which is analogous to the previous method, except that it loads the *dynamic* KB the `solve` method above will operate upon.

- `fun member(Any, Any): Struct` which is an ordinary method aimed at easily creating invocations of the `member/2` built-in predicate. While we only discuss this method, the main idea here is that every standard built-in predicate in Prolog has a Kotlin counterpart in `PrologWithResolution` accepting the proper amount or arguments and returning an invocation to that built-in predicate. This is aimed easing the exploitation of the standard Prolog predicates to developers leveraging our DSL. So, for instance, we also provide facility methods for built-in such as `is/2`, `+/2`, `-/2`, `!/0`, `fail/0`, `append/3`, etc.

Instances of `PrologWithResolution` are created and used via the

```
fun <R> prolog(PrologWithResolution.()-> R): R
```

Listing 11.4: Logic theory aimed at computing solutions for the N-Queens problem

```
no_attack((X1, Y1), (X2, Y2)) :-
  X1 =\= X2, % infix operator
  Y1 =\= Y2,
  (Y2 - Y1) =\= (X2 - X1),
  (Y2 - Y1) =\= (X1 - X2). % arithmetic expression

no_attack_all(_, []).
no_attack_all(C , [H | Hs]) :-
  no_attack(C, H),
  no_attack_all(C, Hs).

solution(_, []).
solution(N, [(X, Y) | Cs]) :-
  solution(N, Cs),
  between(1, N, Y), % built-in predicate
  no_attack_all((X, Y), Cs).
```

static method, which accepts a lambda expression letting the user exploit the DSL on the fly, and returns the object created by this lambda expression. This is for instance what enables developers to write the code snippet discussed in section 11.2.2.

11.3 Case study: N-Queens

We present now a brief example demonstrating how, by integrating multiple programming paradigms, developers may easily produce compact and effective solutions. Suppose one is willing to implement the following Kotlin method

```
fun nQueens(n: Int): Sequence<List<Position>>
```

aimed at computing all the possible solutions to the N-Queens problem. More precisely, the method is expected to enumerate all the possible dispositions of n queens on a $n \times n$ chessboard, such that no queen can be attacked by others. Each solution can be represented by a list of queen positions, in the form $[(1, Y_1), \ldots, (n, Y_n)]$, where each Y_i represent the row occupied by the i^{th} queen—i.e., the one in column i.

Computing all solutions for the N-Queens problem may require a lot of code in most programming paradigms. However, in LP, the solution is straightforward and compact. One may, for instance, leverage the Prolog code depicted in listing 11.4. which can satisfy queries in the form

```
?- solution(N, [(1, Y1), ..., (N, YN)])
```

(provided that some actual N is given) by instantiating each variable `Yi`.

Listing 11.5: Kotlin code aimed generating the theory depicted in listing 11.4

```kotlin
fun nQueens(n: Int) = prolog {
  staticKb(
    rule {
      "no_attack"(("X1" and "Y1"), ("X2" and "Y2")) `if` (
          ("X1" `=!=` "X2") and // infix operator
          ("Y1" `=!=` "Y2") and
          (("Y2" - "Y1") `=!=` ("X2" - "X1")) and
          (("Y2" - "Y1") `=!=` ("X1" - "X2")) // arithmetic expr
        )
    },
    fact { "no_attack_all"(`_`, emptyList) },
    rule {
      "no_attack_all"("C", consOf("H", "Hs")) `if` (
          "no_attack"("C", "H") and
          "no_attack_all"("C", "Hs")
        )
    },
    fact { "solution"(`_`, emptyList) },
    rule {
      "solution"("N", consOf(("X" and "Y"), "Cs")) `if` (
          "solution"("N", "Cs") and
          between(1, "N", "Y") and // built-in predicate
          "no_attack_all"(("X" and "Y"), "Cs")
        )
    }
  )
  return solve("solution"(n, (1 .. n).map { it and "Y$it" }))
}
```

Thanks to our Kotlin DSL for Prolog, one may exploit the elegance of LP in implementing the aforementioned `nQueens` method. For instance, one may implement it as shown in listing 11.5. This implementation produces a *lazy* stream of solutions to the N-Queens problem, given a particular value of n. In doing so, it takes advantages not only of logic- but also of functional-programming paradigm. For instance, on the last line, it exploits the `map` high-order function to build a list in the form $[(1, Y_1),\ \ldots,\ (n, Y_n)]$, to be provided as argument of `solution/2`.

A number of minutiæ may be noted as well by comparing the Prolog code with its Kotlin counterpart. For instance, Prolog operators (such as `=\=/2`, `-/2`, etc.) retain their infix notations in the Prolog DSL. This is possible because they are part of the DSL, in the same way as Prolog built-in predicates (such as `between/3`). This implies the Kotlin compiler can prevent standard operators and built-in from being silently mistyped.

11.4 Recap and Research Perspectives

In this chapter we propose a novel way of integrating the logic, object-oriented, functional, and imperative programming paradigms into a single language, namely

Kotlin. More precisely, we describe how a domain-specific language (DSL) for Prolog can be built on top of the Kotlin language by exploiting the 2P-Kt library. The proposed solution extends the Kotlin language with LP facilities by only relying on its own mechanisms – therefore no external tool is necessary apart from Kotlin itself and 2P-Kt –, even if, however, analogous extensions can in principle be constructed for other high-level languages as well—such as Scala.

Our DSL is currently implemented as part of the 2P-Kt project – namely, within the `dsl-*` modules –, and it is hosted on both GitHub [2P-21] and Maven Central. Other than being available to the public for general purpose usage – under the terms of the Apache License 2.0[7] open-source license –, the DSL is currently extensively exploited to implement the unit tests of 2P-Kt itself.

While in this chapter we discuss the design rationale and architecture of our DSL by explicitly focusing on Kotlin as our target technology, in the future we plan to generalise our approach, possibly tending to technology independence. Furthermore, we plan to provide other implementations of our DSL, targeting other high-level languages and platforms, in order to make logic programming and its valuable features available in general-purpose programming languages and frameworks.

As concerns possible research directions and applications, in the future, we plan to exploit our DSL in several contexts. For instance, we argue our DSL may ease the usage of the logic tuple spaces offered by the TuSoW technology [CROM19]. Similarly, we believe our DSL may be used as a basic brick in the creation of logic-based technologies for MAS, as the MAS community is eager of general-purpose, logic-based technologies targetting the JVM platform [CCMO21a]. Along this line, we argue logic-based languages for MAS may benefit from the integration of our DSL – or a similar one – to support the reasoning capabilities of agents. Finally, we plan to exploit our DSL within the scope of XAI [CCO20], to ease the creation of hybrid (i.e., logic + machine-learning) systems, and management on the symbolic side.

[7]https://www.apache.org/licenses/LICENSE-2.0

Chapter 12

Bridging LP and Machine Learning

This chapter contains contributions from the following Master's thesis: [Cas21], which we supervised

Symbolic and sub-symbolic AI are complementary under a number of perspectives. The same is true for their major techniques, namely logic programming (LP) and neural networks (NN). For this reason, many recent contributions from the literature are discussing the possible frameworks for their integration and hybridisation.

However, what is currently slowing down scientific progress in this context is not the lack of ideas concerning *how* such integration and hybridisation may occur. Rather, we believe the bottleneck is caused by the lack of suitable technologies enabling and easing the experimentation of integrated or hybrid systems. Logic-based technologies are in fact technological islands, for which poor care is given to the construction of bridges with the rest of the AI land.

Accordingly, in this chapter, we address the issue of supporting machine learning (ML) – and, in particular, neural-networks-based training and inference – in logic programming. In doing so, we follow the twofold purpose of

1. letting logic programmers exploit the benefits of sub-symbolic AI, and, in particular, neural networks; and

2. enabling the experimentation of hybrid systems – involving both logic and neural processing of data – in practice.

More precisely, we discuss the design and prototyping of a logic based API for machine learning. Such API consists of a set of logic predicates enabling the representation, training, testing, and exploitation of sub-symbolic predictors in LP—possibly, out of data expressed in logic form. In other words, our API lets logic programmers use neural networks in their programs – e.g. to train or exploit classifiers or regressors –, without requiring them to abandon the logic realm. Of course, to make this possible, our API supports the whole gamma of low level tasks which are commonly involved in a ML workflow—including, but not limited to, data preprocessing, cross-validation, etc.

Technically, we prototype our API via a logic library – namely, the ML-Lib – targetting the 2P-Kt ecosystem, and the JVM platform. It consists of a number of primitives bridging the logic realm with some underlying machine-learning library, allowing Prolog to manipulate sub-symbolic facilities such as datasets, neurons, layers, or activation functions, and vice versa. DeepLearning4J is the underlying library we leverage upon in this chapter—selected after meticulous technological analysis (here reported as well). However, our design is general enough to support other libraries and, possibly, different platforms—e.g. Tensorflow over Python.

In perspective, we argue that our work represents the first step towards a wider degree of interoperability among logic- and sub-symbolic AI. In fact, one the long run, we aim to enable the design and construction of hybrid systems, fruitfully and dynamically combining the major advantages of both approaches to artificial intelligence by mixing inferential (via logic programming) and intuitive (via neural networks) reasoning capabilities. Along this path, the proposed API is one key enabling factor, as it supports the creation of logic-based inferential engines which are capable of learning from data via state-of-the-art mechanisms. Dually, by supporting the training of neural networks from logic data, our API can also be considered as a tool for endowing sub-symbolic predictors with prior, high-level knowledge.

Accordingly, the reminder of this chapter is organised as follows. In section 12.1, we describe and analyse the supervised ML domain, with the purpose of identifying which functionalities our logic API for ML should support, and how exactly they are expected to work. Then, in section 12.2, we delve into the design of the ML-Lib discussing and motivating conventions, syntactic choices, and architectural decisions. In section 12.3, we then discuss a number of technology- and platform-related aspects arising when prototyping the ML-Lib. There, we motivate technological commitments such as 2P-Kt and DeepLearning4J. Section 12.4 is where we discuss a number of examples concerning the usage of the ML-Lib, and its potential applications. Finally, section 12.5 concludes the chapter, providing some insights about the future research directions stemming from this work.

12.1 Logic API for ML: Requirements, Analysis, and Modelling

12.1.1 Goals

Broadly speaking, a logic API for ML enables the combination or integration of symbolic AI's expressiveness and sub-symbolic AI's flexibility, at a deeper and unprecedented level. Here we describe a number of motivating features, which we choose to pursue as goals.

Declarative machine learning. Declarative ML is a paradigm by which data scientists' code should only specify *what* a ML workflow should do, by leaving the underlying platform in charge to understand *how*. This is partially supported by the current practice of data science which relies on high-level languages (like Python) and libraries of elementary components to be composed (e.g. Scikit-Learn). However, the solutions proposed so far do not leverage upon inherently declarative frameworks like LP, but rather on object oriented languages—which eventually need imperative statements to be provided by data scientists. An API for ML should then be designed to support the declarative expression of all possible aspects of a ML workflow in the LP framework. This would be useful under both a logic and ML perspective. In fact, it would pave the way towards the exploitation of ML within the LP community, other than providing data scientists with a way to describe their ML workflows in a formal and runnable way.

Symbolic data sources. Logic knowledge bases are a peculiar way of collecting knowledge. Unlike datasets and DBMS, they represent information in symbolic form, via – possibly *intensional* – logic formulæ. Hence, they can virtually represent any sort of datum – be it atomic, compound or structured – via a concise (yet very expressive) language, while possibly saving space. Accordingly, when combining LP with ML, knowledge bases should be exploitable as data sources as well—other than ordinary CSV files or relational databases.

Hybrid reasoning. Automatic reasoning may greatly benefit from sub-symbolic AI to overcome its inherent crispness. Fuzzy data could then be suitably and coherently processed by a sub-symbolic predictor as part of a wider symbolic resolution process. To make this possible, sub-symbolic predictors should be representable, trainable, and queryable as any other logic predicate, without requiring the semantics of logic resolution to be affected. Consequently, logic programs should be endowed with ad-hoc predicates and syntactical categories, aimed at representing and manipulating sub-symbolic predictors and data.

Model selection via resolution. Logic resolution essentially consist of a search procedure aimed at finding solutions in a proof tree. This could be applied to a common step of any ML workflow—namely *model selection*. There, data scientists must assess several predictor families, to select the one which is better suited for the learning task at hand. Then, they must search for the best hyper-parameters for the selected family of predictors. All such choices involve several sorts of predictors, with possibly different hyper-parameters, to be trained and compared—either in an orderly fashion or in parallel. LP naturally captures the non-deterministic exploration of a space of possible choices. Hence it is well suited to both declaratively represent and implement model selection.

12.1.2 Domain Description

To support the aforementioned goals, our logic API must cover the full gamma of tasks involved in any possible ML workflow.

Briefly speaking, a ML workflow is the process of producing a suitable predictor for the available data and the learning task at hand, following the purpose of later exploiting that predictor to draw analyses or to drive decisions. Hence, any ML workflow is commonly described as composed by two major phases, namely training – where predictors are fit on data – and inference—where predictors are exploited. However, in practice, further phases are included, such as data provisioning and preprocessing, as well as model selection and assessment.

In other words, before using a neural network in a real-world scenario, data scientists must ensure it has been sufficiently trained and its predictive performance is sufficiently high. In turn, training requires *(i)* an adequate amount of data to be available, *(ii)* the structure of the network under training to be defined, and *(iii)* any other hyper-parameter (e.g. learning rate, momentum, batch size, epoch limit, etc.) to be fixed. Data must therefore be provisioned before training, and, possibly, pre-processed to ease training it self—e.g. by normalising data or by encoding non-numeric features into numeric form. The structure of the network must be defined in terms of (roughly) input, hidden, and output layers, as well as their activation functions. Finally, hyper parameters must be carefully tuned according to the data scientist's experience, and the time constraints and computational resources at hand.

Thus, from a coarse-grained perspective, a machine learning workflow can be conceived as composed by six major phases, overviewed in fig. 12.1 and described into the following paragraphs. The background colour of each phase denotes the expected design and implementation effort for supporting that phase in our logic API for ML (low effort $\leftrightarrow$ green, moderate effort $\leftrightarrow$ yellow, considerable effort $\leftrightarrow$ red).

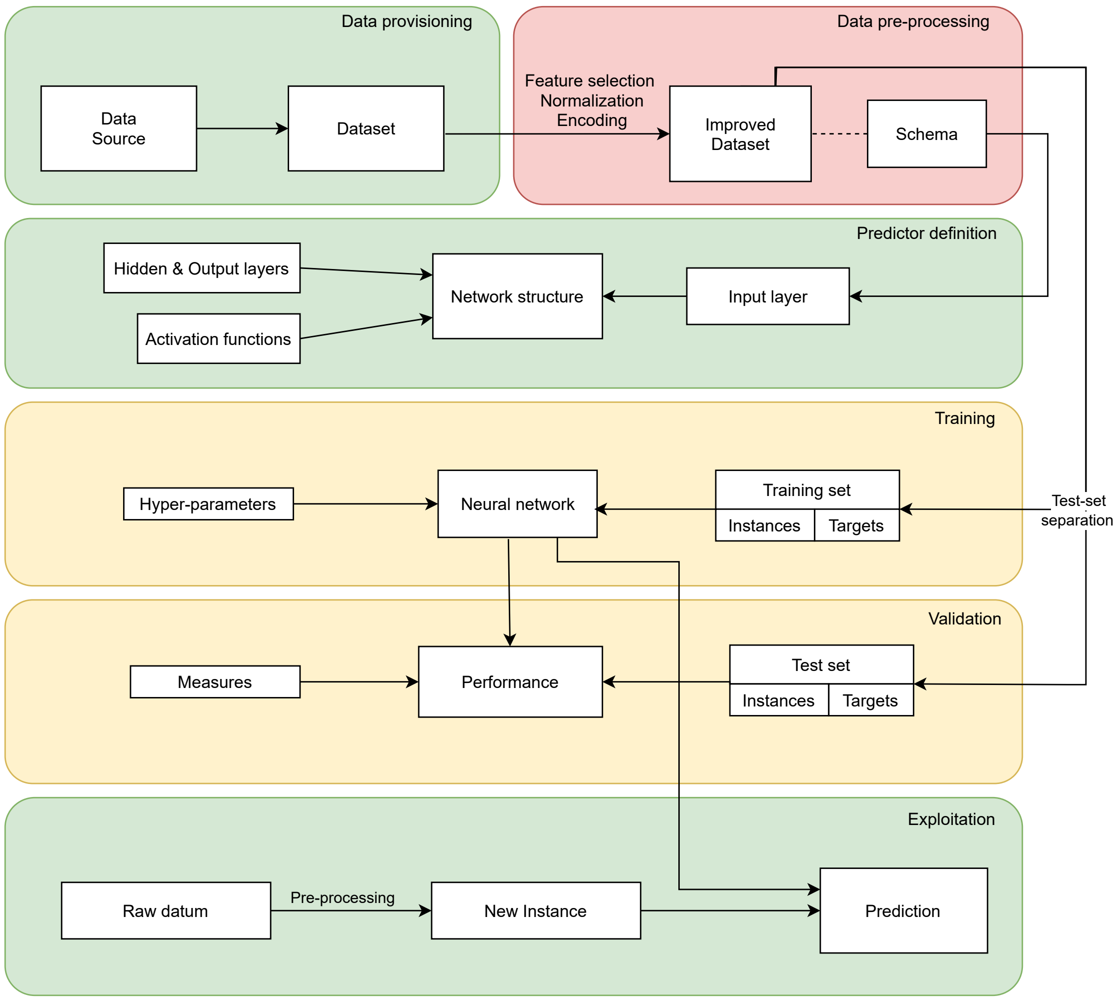


Figure 12.1: Major phases of the generic ML workflow

Dataset loading. Provided that the data provisioning phase has resulted into a dataset – roughly, a collection of homogenous data, often coming in the form of a single file, a folder or a database –, the first step of any ML workflow consists of loading that dataset in memory for later processing. To support such step, ML frameworks come with ad-hoc functionalities aimed at loading the dataset by reading a file from the local file system, by fetching it from the Web, or by querying a DBMS. These usually come in the form of either classes or functions, coherently w.r.t. the object-oriented nature of mainstream ML frameworks.

Our logic API for ML should expose ad-hoc *predicates* to serve the same purposes. Furthermore, however, it should also support the loading of datasets out of logic theories of facts and rules.

Data pre-processing. Raw datasets are often inadequate to favour predictors' training. Hence, dataset pre-processing is commonly practised to increase the effectiveness of any subsequent training phase. Pre-processing, in practice, involves a number of bulk operations to be applied to the whole dataset, following several purposes, such as: *(i)* homogenize the variation ranges of the many features sampled by the dataset, *(ii)* detect irrelevant features and remove them, *(iii)* construct relevant features by combining the existing ones, *(iv)* encoding non-numeric features into numeric form, and *(v)* horizontal (by row) or vertical (by column) partitioning of the dataset. Purpose *(v)*, in particular, is of paramount importance, as it supports the *test set separation* – that is, a fundamental step to be performed at the very beginning of any correct ML workflow, to later enable validation and testing –, as well as splitting input-related columns from output-related ones—which, in turn, is fundamental to support training.

Notably, our logic API for ML should support all such purposes, via a concise (yet expressive) predicates letting the logic data-scientist decide which preprocessing operations to perform, and when. In practice, this involves a number of predicates supporting bulk manipulations to be applied to the whole dataset, such as computing statistical moments (e.g. mean, variance, standard deviation, etc.), or aggregated measures (e.g. min, max, etc.) on a column-wise basis, as well as transforming (e.g. transforming a categorical feature in numeric form via the one-hot encoding).

Predictor selection and definition. Many sorts of predictors could be used in principle to perform supervised learning—e.g. neural networks, decision trees, support vector machines, etc. Unless some technical or administrative constraint exists, it is a common practice for data scientists to spend some time selecting the most adequate sort of predictor for the data and the learning task at hand. This is a common phase in virtually any ML workflow. Once a particular sort of predictor has been chosen, data scientists need a way to specify the shape the to-be-trained predictor should have. Of course, such specification should take into account the schema of the input data, as well as the schema of the expected outcomes to be produced by the predictor. The are however other aspects to be tuned, generally referred to as *hyper-parameters*.

Consider the case of neural networks as an example. Decision points in this case concern the choice of *(i)* which and how many hidden layers (of neurons) to adopt, *(ii)* how to interconnect them, and *(iii)* which activation functions to adopt for the neurons therein contained.

Accordingly, our logic API for ML should support the specification of as many sorts of predictors as possible, as well as their parametrisation. Once again, predicates should be defined to serve this purpose. In particular, at least one ad-hoc predicate should be defined for each sort of predictor to be supported, carrying as many arguments as the possible hyper-parameters that could be specified for that sort of predictors. In case hyper-parameters cannot conveniently represented as raw logic types (numbers or strings), ad-hoc predicates should be provided as well for constructing structured hyper-parameters values.

In the particular case of neural networks, ad-hoc predicates should be provided to construct layers, and activation functions, and to combine them to create arbitrarily complex network architectures.

Training. Predictors' training plays a pivotal role in ML workflows. This is the phase where predictors are fit on the available data or, in other words, automated learning actually occurs. Generally speaking, training can be modelled in LP as single predicate, mapping untrained predictors into trained ones, possibly via a number of learning parameters (e.g. learning rate or momentum for NN, or maximum depth for DT), or stopping criteria (e.g. max epochs for NN, or max depth for DT), other than, of course, the data to be used for training. Once again, several ad-hoc predicates should be defined to support structured parameters or stopping criteria in our logic API for ML. Furthermore, regardless of its shape, the training predicate should accept some arguments aimed at specifying whether the columns of the training set should be considered as inputs or outputs.

Exploitation. Exploitation is commonly the last phase of any ML workflow. Here, trained predictors are used to draw predictions on new data—i.e. different data w.r.t. the one used for training. In particular, given a raw datum having the same schema of the input data used for training – there including any prior pre-processing phase –, the trained predictor can be exploited to compute the corresponding prediction—even if (and especially because) the raw datum has never been observed before by that predictor. In most common cases, predictions attempt to solve classification or regression problems. In any case, yet another general predicate should be added to our logic API for ML to support drawing predictions out of a trained predictor and a set of raw data (or a single datum). Ad-hoc predicates may be provided as well to explicitly model higher-level tasks, such as classification and regression. Finally, it should be possible to store, retrieve, and re-apply any pre-processing procedure possibly defined before training, to the raw data for which predictions should be drawn—in order to make it acceptable for the predictor as an input.

Validation. Validation is the *penultimate* step of any ML workflow: it succeeds training and precedes exploitation. It is here discussed as last because it technically relies on the capability of drawing predictions via trained predictors—which is treated in the paragraph above.

Generally speaking, validation attempts to measure the predictive performance of a trained predictor, with the purpose of assessing if and to what extent it will generalise to new, unseen data. To do so, the predictor is tested against the test set—that is, a collection of unseen data, for each expected predictions exist. The discrepancy (or similarity) among the actual and expected predictions is then measured via ad-hoc scoring functions (a.k.a. measures), resulting in a performance assessment for the trained predictor. Many measures may be used to assess classifiers (e.g. accuracy, F1-score, etc.) and as many to assess regressors (e.g. MAE, MSE, R^2, etc.). Hence, to support validation, our logic API for ML should provide predicates to compute each possible measure.

12.1.3 Analysis and Modelling

Here we analyse the ML domain w.r.t. our goals, and we elicit the most relevant entities and actions our logic API should support. In other words, we derive the meta-model leading the design, implementation, and usage of our logic API for ML.

There are five major sorts of entities by which any ML workflow can be described. These are introduced below:

Value: a scalar, vectorial, matrix, or tensorial datum from a given domain (e.g. an integer or real number, or a vectors of integer or real numbers, as well as a string, a table, a time series, etc.).

Schema: a concise and formal description of a domain (i.e. a set of values). For scalar values, schemas are essentially data types (e.g. integers, reals, strings, etc.), while for non-scalar data they carry information about the name, index, and type of each single scalar component.

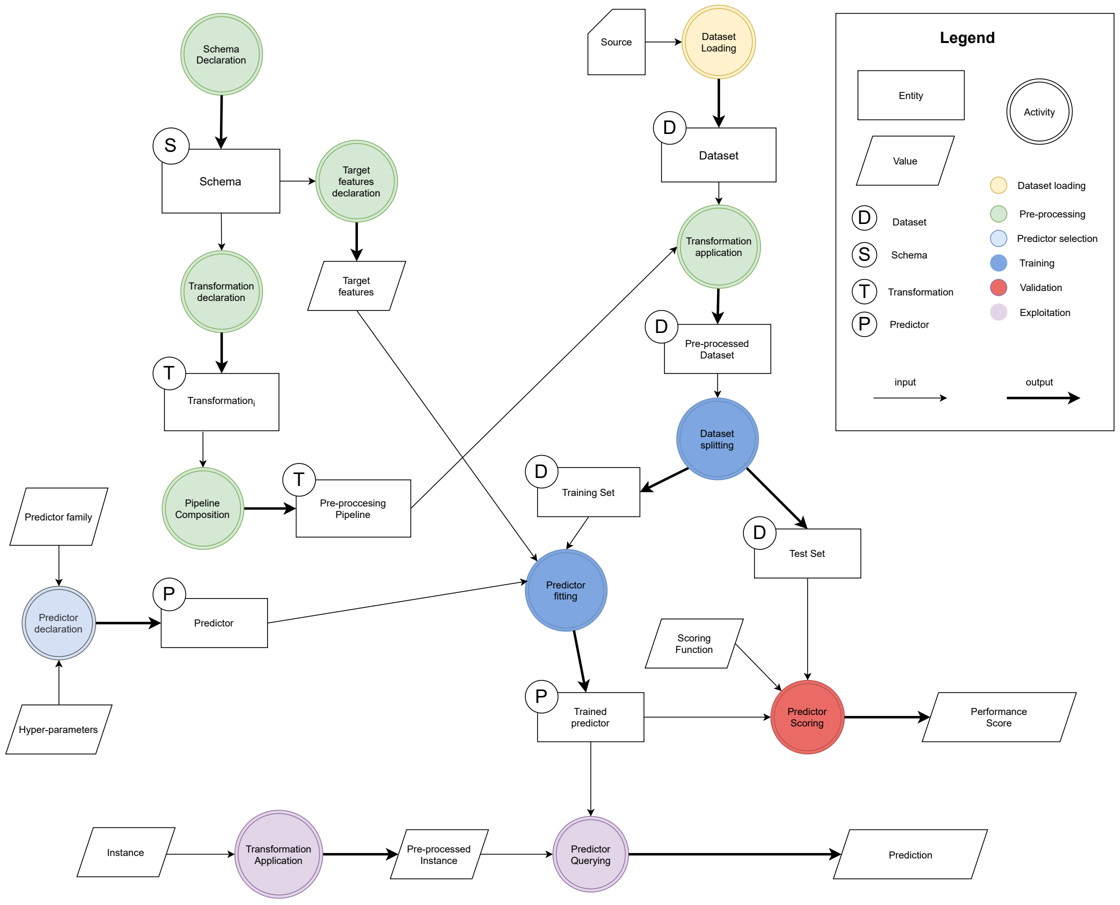


Figure 12.2: Model and meta-model of a logic API for ML

Dataset: a collection of values matching a particular schema—which is supposed to be known.

Transformation: any operation aimed at transforming an entity dataset into another other—commonly, a dataset into either another dataset (e.g. normalization, standardization, etc.) or a value (e.g. max, min, average, etc.) From an algebraic perspective, it is a function. From a computational perspective it is an algorithm.

Predictor: a stateful computational entity capable of *(i)* drawing predictions (i.e. outputting values) out of (possibly unseen) input values, according to its internal state *(ii)* updating its internal state according to a dataset (to improve future predictions)

Our logic API for ML supports the representation, combination, and manipulation of entities of these kinds. In particular, fig. 12.2 depicts the overall workflow, the specific activities therein contained, and the involved entities, from the user perspective. Each phase of a ML workflow is characterised by a specific set of activities the data scientist may be willing to perform. These should be therefore declaratively representable in logic.

Accordingly, in the reminder of this section, we enumerate the most relevant activities our logic API should support, and the entities they operate upon. Notably, activities are grouped w.r.t. the ML phases they operate in, according to the ML workflow elicited in section 12.1.2.

For instance, the following activities make sense into the *dataset loading* phase as their major purpose it to support the loading of a dataset into a solver's memory, and its preparation for sub-sequent processing:

Dataset loading — i.e. the operation of loading a dataset from either a value – representing either a local or remote file –, or from a Prolog theory

Schema declaration — i.e. the operation of constructing a representation for a given schema

Target features declaration — i.e. the operation of tagging a portion of features of some schema as either inputs or outputs (a.k.a. targets)

Dataset splitting — i.e. the operation of horizontally partitioning a dataset into two or more smaller datasets

Subsequently, data scientists will commonly enter the *dataset pre-processing* phase. Here, they may be willing to define transformations or cascades of transformations (pipelines, henceforth) to be eventually applied on datasets:

Transformation declaration — i.e. the operation of declaratively encoding a transformation operation to be applied to all data in a dataset Such kinds of transformations can be modelled as functions accepting a dataset as input and producing a dataset as output

Pipeline composition — i.e. the operation of declaratively constructing a composite transformation as a cascade of simpler transformations

Transfornmation application to a dataset — i.e. the operation of actually constructing a new dataset from a prior dataset and a transformation

The next phase commonly involves the *definition* of one or more predictors via a unique meta-activity, namely:

Predictor declaration — i.e. the operation of constructing a representation for a particular predictor, which implies choosing the predictor family and specifying actual values for its hyper parameters

Of course, as many families of predictors exist, and each one is characterised by its own set of formal hyper-parameters, many variant if this operation may eventually be defined in practice.

Eventually, declared predictors may enter the *training* phase, meaning that their learning from data should be triggered. This can be achieved via yet another activity, namely:

Predictor fitting w.r.t. a training set of data — i.e. the operation of fitting a predictors' internal parameter on some provided training data

Again, given the variety of predictors available, this operation may come with several predictor specific variants accepting different learning parameters.

Once in their *inference* phase, trained predictors may eventually be exploited to be draw predictions. Hence, this is yet another relevant activity, namely:

Predictor querying — i.e. the operation where (possibly unseen) values are provided to some trained predictor as a query, and the resulting values are interpreted as predictions

Finally, in the *validation* phase, trained predictors should be assessed by measuring their performance w.r.t. some test data This is yet another meta-activity, with several possible variants depending on the particular measure being exploited:

Predictor scoring — i.e. the operation of computing a scoring value out of a trained predictor, a test dataset, and a scoring function

Figure 12.2 provides an overview of the overall workflow these activities are involved into. In particular, it represents inter-dependencies among all such activities, other than stressing what sorts of entities they accept as input and produce as output. In the next section, we discuss how these activities are reified into actual actual predicates.

12.2 Realising the API: ML-Lib Design

Here we discuss the design of ML-Lib, i.e. a logic programming library reifying the logic API for ML modelled in section 12.1.

The overall architecture is depicted in fig. 12.3. The ML-Lib assumes a goal-oriented logic solver being in place, where ordinary logic programs can be executed. Thanks to the ML-Lib, these logic programs may also exploit a number of predicates aimed at training and using ML predictors—other than any other entity involved in the process. Behind the scenes, the library also assumes an underlying object-oriented (OO) library providing high-level ML abstractions, such as

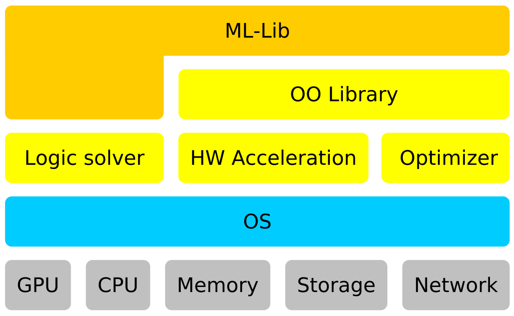


Figure 12.3: Layered view of our ML-Lib. An OO library is assumed behind the scenes, providing high-level abstraction to optimize ML predictors, possibly via HW acceleration

datasets, predictors, and so on. Examples of these libraries may be for instance Keras or DeepLearning4J. The OO library may in turn be backed by an optimizer, i.e. a low-level software taking care of making training and data management effective on the available hardware—and possibly exploiting hardware acceleration (via GPU) to the purpose. In practice, software such as Theano, Caffe, or Tensorflow may serve this purpose. Actual technological choices may finally depend on the particular runtime platforms being targeted. For instance, targeting the JVM may imply DeepLearning4J must be exploited behind the scenes, while targetting Python may pave the way towards the exploitation of both Keras and Tensorflow. However, while technological choices are contingent and subject to change, the overall architecture is meant to support the implementation of the ML-Lib as a façade towards the underlying OO library, regardless of what it is.

At the functional level, the design of our ML-Lib is provided in terms the entities from section 12.1, and the logic predicates available to create, manipulate, or represent them. Figure 12.4 provides an overview of these predicates, grouped by entities. Both in the figure and in the reminder of this section, we adopt the following notation to denote the interfaces of logic predicates:

$$\texttt{functor(}\odot_1\ \texttt{Name}_1\texttt{:}\ \ \textit{\texttt{type}}_1\texttt{, ..., }\odot_N\ \texttt{Name}_N\texttt{:}\ \ \textit{\texttt{type}}_N\texttt{)}$$

where N denotes the arity of predicate $\texttt{functor/}N$, whose i^{th} argument – named $\texttt{Name}_i$ – must be of type $\textit{\texttt{type}}_i$, and it must be considered as an input or output parameter depending on the mode indicator[1]$\odot_i$. So, for instance, we denote input parameters by `+`, output parameters by `-`, and input-output parameters by `?`. Admissible arguments types include constant term types (`integer`, `real`, `atom`), structured term types (`compound`, `list`), as well as *references* (`ref`), and

[1] `cf.https://www.swi-prolog.org/pldoc/man?section=preddesc`

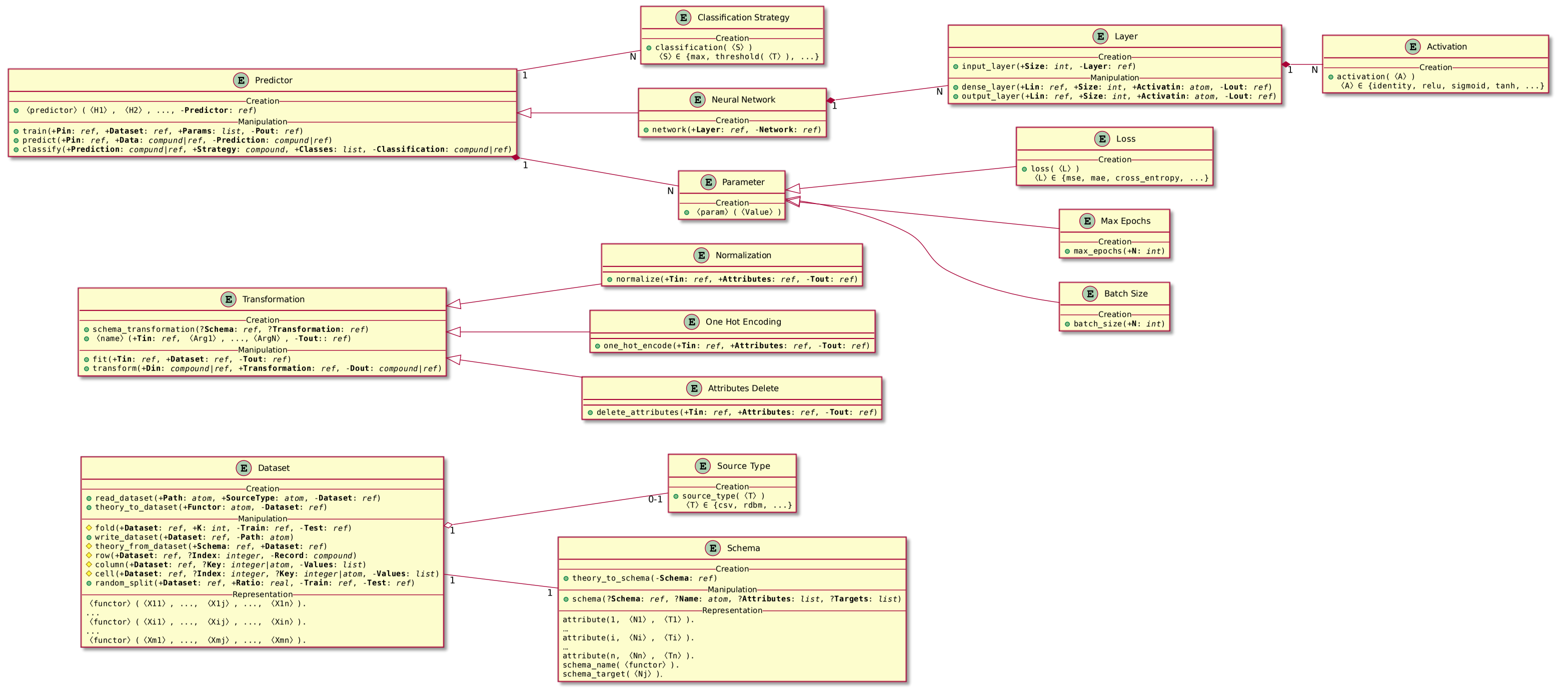


Figure 12.4: Overview of our ML-Lib's design. The chart represents the many entities logic programmers may exploit via our ML-Lib, and the many predicates supporting their creation, manipulation, or representation. Predicates are depicted with either a yellow diamond in case they are non-deterministic (a.k.a. backtrackable), or a green circle otherwise.

union types ($\texttt{T}_1\texttt{|T}_2\texttt{|...}$). References, in particular, are a special kind of constant term, whose instances represent objects from the object-oriented realm. These are necessary to make our ML-Lib able to operate with the non-logic entities exposed by the underlying OO library supporting ML.

Accordingly, in the reminder of this section, we enumerate the predicates constituting our ML-Lib, categorised w.r.t. the entities they act upon. In particular, the ML-Lib exposes predicates covering 4 major sorts of entities – i.e. the ones elicited in section 12.1.3, namely: Schema, Dataset, Transformation, and Predictor –, plus a number of ancillary entities aimed at supporting their manipulation – such as Classification Strategy, Source Type, and Parameter – or specialising their behaviour—such as Neural Network, and Layer.

12.2.1 Schemas

Schemas are concise metadata describing datasets' columns. They define their indexes, names, and admissible types, and they are assumed to be declared by the user.

The ML-Lib supports schemas represented as any of two forms: either as clauses or as objects—to be represented in LP via reference terms. Ad-hoc predicates are provided to support the conversion from one form to the other.

Schemas as clauses. In the general case, schema declarations are firstly provided by the user in clausal form. This requires the user to fill the logic theory with clauses of the form:

$$\begin{array}{l}
\texttt{attribute(1, } N_1\texttt{, } T_1\texttt{).}\\
\quad\vdots\\
\texttt{attribute(}i\texttt{, } N_i\texttt{, } T_i\texttt{).}\\
\quad\vdots\\
\texttt{attribute(}n\texttt{, } N_n\texttt{, } T_n\texttt{).}\\
\texttt{schema\_name(}N\texttt{).}\\
\texttt{schema\_targets([}N_j\texttt{, } N_k\texttt{, ..., } N_h\texttt{]).}
\end{array}$$

where N is the name of the schema, and n is the total amount of attributes declared for that schema, while N_i is the name of the i^{th} attribute, and T_i is its type. Indexes $j, k, h \in \{1, \ldots, n\}$ aim at selecting attributes names declared as *targets*—i.e. as outputs of the learning process. While attribute (N_i) and schema (N) names are simple atoms, attribute types (T_i) are compound terms for which the `attribute_type(`T_i`)` holds true.

The `attribute_type/1` predicate is defined as follows:

```
attribute_type(string).
attribute_type(integer).
attribute_type(real).
attribute_type(boolean).
attribute_type(categorical([_ | _])).
attribute_type(ordinal([_ | _])).
```

Hence, admissible attribute types involve infinite domains such as the numeric (either integer or real numbers), and strings ones, as well as finite domains such as booleans, and categorical (i.e. unordered) or ordinal sets of constant values.

Schemas as objects. To be exploitable by the underlying OO library, schemas must be represented as objects. Schemas represented in clausal form can be converted into object form via the following predicate:

```
theory_to_schema(-Schema: ref)
```

which *(i)* inspects the current KB looking for a schema description in clausal form, *(ii)* instantiates a new schema object in the underlying OO library, *(iii)* creates a new reference term referencing the newly created schema, *(iv)* unifies that term with the output parameter denoted by `Schema`.

References to schemas in object form may be then passed as arguments to many other predicates from the ML-Lib in order to provide them the necessary metadata to manipulate datasets.

Manipulating schemas. A part from schema declaration or creation, other relevant operations over schemas involve the inspection (i.e. reading) of their components—namely, names, attribute names, attribute types, and targets. This can be achieved via the following predicate:

```
schema(?Schema: ref, ?Name: atom, ?Attributes: list, ?Targets:
                                  list)
```

Given a schema reference, the predicate retrieves *(i)* the schema's name, which is unified with `Name`, *(ii)* the list schema attributes – where each attribute has the form `attribute(`i`, `N_i`, `T_i`)` –, which is unified with `Attributes`, and *(iii)* the list of schema targets – where each target is an atom acting as attribute name –, which is unified with `Targets`. Notably, the predicate is *bi-directional* and its arguments can act as either input or output parameters. In case an unbound `Schema` variable is provided as output parameter, and assuming that the `Name`, `Attributes`, and `Targets` parameters are fully instantiated, the `schema/4` predicate acts as yet another way to create a schema in object form—and the newly created schema is bound to `Schema`.

12.2.2 Datasets

A dataset is a tabular representation of a bunch of homogenous data records. As such, a dataset is characterised by a schema and a number of records matching that schema.

Similarly to what it does for schemas, the ML-Lib supports datasets represented as either clauses or objects. Ad-hoc predicates are provided to support the conversion from one form to the other, other than for loading datasets from some data source, such as a file or a DBMS.

Datasets as objects. In the general case, datasets objects are firstly loaded from a data source. These may be local or remote files – commonly in "comma separated values" (CSV) format –, as well as DBMS of any sort—provided that adequate connection support is provided by the underlying OO library, or any other third-party module. The ML-Lib provides a unique entry point to load a dataset from any data source, namely:

```
read_dataset(+Location: atom, +SourceType: atom, -Dataset: ref)
```

This predicate aims at loading the dataset from a given `Location`—be it a path on the local filesystem, a URL referencing some remote resource, or a connection string for some DBMS. It also requires the caller to specify the `SourceType` the dataset should be read from. Regardless of the particular location and source type, the behaviour of the `read_dataset/3` predicate is such that: *(i)* raw data is retrieved from `Location`, and *(ii)* parsed according to the selected source `SourceType`; finally *(iii)* a new dataset object is created along with a reference term for it, *(iv)* which is then unified with `Dataset`.

Admissible values for the `SourceType` parameter are determined by the `source_type/1` predicate, defined as follows:

```
source_type(csv).
```

meaning that currently the ML-Lib only supports data provisioning from CSV files. However, further source types are going be supported in the future. That will imply extending the `source_type/1` predicate definition with further cases.

Datasets as clauses. Logic programmers may also be willing to describe the dataset via a logic theory. When this is the case, the theory should contain not only the clauses describing the schema (i.e. the dataset's columns), but also a number of clauses describing the actual content of the dataset (i.e. its rows). In particular, the ML-Lib expects data entries to be provided as clauses of the form:

$$
\begin{array}{c}
N\texttt{(X}_{1,1}\texttt{, ..., X}_{1,j}\texttt{, ..., X}_{1,n}\texttt{).}\\
\vdots\\
N\texttt{(X}_{i,1}\texttt{, ..., X}_{i,j}\texttt{, ..., X}_{i,n}\texttt{).}\\
\vdots\\
N\texttt{(X}_{m,1}\texttt{, ..., X}_{m,j}\texttt{, ..., X}_{m,n}\texttt{).}
\end{array}
$$

where N is the schema name declared via `schema_name/1`, and $\mathtt{X}_{i,j}$ is the value of the j^{th} attribute of the i^{th} data entry. Of course, the actual type of $\mathtt{X}_{i,j}$ must be coherent with the formal type T_i declared in the schema definition.

Datasets in clausal form must be converted into object form to be exploitable by the underlying OO library. This can be achieved via the following predicate:

```
theory_to_dataset(+SchemaName: atom, -Dataset: ref)
```

which *(i)* inspects the current KB looking for one or clauses using `SchemaName` as the head functor, *(ii)* instantiates a new dataset object in the underlying OO library, *(iii)* populates it with as many rows as the aforementioned clauses, *(iv)* creates a new reference term referencing the newly created dataset, *(v)* unifies that term with the output parameter denoted by `Dataset`. Of course, this predicate also takes into account the schema-related metadata which are assumed to be defined in clausal form as well.

Datasets manipulation. Datasets are amongst the basic bricks of predictors training in ML, hence they must support several kinds of manipulations. Within the scope of the ML-Lib, we support partitioning a dataset in several ways to support both cross validation and test set separation, other than accessing a dataset by row, column, or cell. Conversions from and into clausal form complete the picture.

Splitting. To support test set separation, the ML-Lib provides a predicate to randomly split a dataset into a training and test set, given a ratio:

```
random_split(+Dataset: ref, +Ratio: real, -Train: ref, -Test:
                              ref)
```

Given a reference to a `Dataset` in object form, and a `Ratio` – i.e. a real number in the range $]0, 1[$ –, the predicate *(i)* randomly samples the given percentage of data entries from `Dataset`, *(ii)* collects them into a new dataset, whose reference is bound to `Test`, and *(iii)* collects the remaining data entries into yet another dataset, whose reference is bound to `Train`. So, for instance, a ratio of 0.1 would randomly split the dataset into a training set containing 90% of the original data, and a test set containing 10% of the original data.

To support cross validation, ML-Lib provides an *ad-hoc* predicate:

`fold(+Dataset: `*`ref`*`, +K: `*`integer`*`, -Train: `*`ref`*`, -Validation: `*`ref`*`)`

which splits the `Dataset` into 2 partitions, namely `Train` and `Validation`, the former containing $\frac{k-1}{k}\%$ data entries – to be used as the training set –, and the latter containing the remaining $\frac{1}{k}\%$ data entries—to be used as the validation set. Both `Train` and `Validation` are bound to reference terms, referencing datasets in object form. Notably, the `fold/2` is non-deterministic as it enumerates all possible folds of a `K`-fold cross validation process. Hence, provided that `K` ≥ 2, the predicate computes `K` partitioning of the original dataset.

Data access. The ML-Lib supports accessing the information encapsulated into a dataset in object form via three predicates, namely:

```
row(+Dataset: ref, ?Index: integer, -Values: list).
column(+Dataset: ref, ?Attribute: integer|atom, -Values: list).
cell(+Dataset: ref, ?Index: integer, ?Attribute: integer|atom,
                          -Values: list).
```

They all are non-deterministic, and they both support the retrieval of a particular row / column / cell from the dataset as well as the enumeration of all possible rows / columns / cells from that dataset.

In particular, predicate `row/3` aims at retrieving rows. If the `Index` parameter is a positive integer, then the predicate attempts to unify the `Value` parameter with the list of values contained the `Index`th row of the dataset. Otherwise, if `Index` is uninstantiated, the predicate enumerates all rows in the dataset, and for each row it unifies the `Index` and `Values` parameters accordingly.

The predicate `column/3` is totally analogous to `row/3`, expect it aims at retrieving or enumerating columns. The only notable difference w.r.t. `row/3` is that columns can be referenced by either attribute names or indexes—thus both positive integers and atoms can be bound to the `Attribute` parameter.

Finally, predicate `cell/4` supports accessing or enumerating cells. In particular, it allows the user to access the `Value` in position (`Index`, `Attribute`), where `Index` is a row index in and `Attribute` is an attribute name or index. If one or both parameters are uninstantiated, the predicate enumerates all possible assignments.

Object to clausal form conversion. The logic programmer may also be willing to convert a dataset in object form into a dataset in clausal form. This can be attained via the following predicate:

`theory_from_dataset(+Schema: `*`ref`*`, +Dataset: `*`ref`*`)`

Given the references to both a dataset and its schema in object form, the predicate populates the solver's *dynamic* KB with the a number of clauses representing the dataset and its schema in the clausal form described above.

12.2.3 Transformations

A transformation is a function altering a dataset and, possibly, its schema. It may be parametric and hence tuned according to the content of the dataset or its schema.

Consider for instance the case of the "Normalization" transformation. It applies an affine transformation to each column of the dataset (independently) in such a way that it has a predefined mean (e.g. 0) and standard deviation (e.g. 1). Hence, it alters the content of a dataset leaving its schema unaffected. To work properly, it requires two major computational steps, namely *(i)* computing (and storing) the mean and standard deviation of each column of the original dataset, *(ii)* applying the affine transformation to normalize the dataset columns (i.e. subtracting the mean and dividing by the standard deviation each cell of each column).

In the general case, transformations are modelled as *stateful* entities supporting at least 2 operations, namely *fitting* and *transforming* a dataset and its schema. The latter operation is also known as "applying a transformation to a dataset", and it should not only support the retrieval of the transformed dataset, but the transformed schema as well. Furthermore, transformations should be composable into *pipelines*, i.e. cascades of simpler transformations to be fitted or applied in a row.

To support all such aspects, the ML-Lib provides predicates aiming to

1. create a transformation given a schema,
2. combine elementary transformations into composite transformations,
3. fit transformations over data (regardless of whether they are elementary or composite),
4. apply composite or elementary transformation to a dataset, thus attaining a new dataset,
5. retrieve the new schema resulting from a transformation application.

Differently from schemas and datasets, for which the ML-Lib supports both clausal and object representations, transformations are only representable in object form, hence the following predicates assume transformations to be manipulated via reference terms.

Transformations to/from schemas. To support aims 1 and 5, the ML-Lib provides the following *bi-directional* predicate:

```
schema_transformation(?Schema: ref, ?Transformation: ref)
```

which changes its behaviour depending on which arguments are instantiated.

In particular, if `Schema` is bound to a schema object, then `Transformation` is unified with an identity transformation – i.e. a transformation leaving the schema and the dataset unaffected –, which can be used as the initial step of a composite pipeline. This is how aim 1 is served.

Conversely, if `Transformation` is bound to an actual transformation object, then `Schema` is unified with the new schema object attained by applying that transformation to the schema it was originally constructed from. This is how aim 5 is served.

Creating and combining elementary transformations. To support aim 2, the ML-Lib provides a number of predicates sharing a similar syntax. Each predicate is in charge of creating a composite transformation by appending a specific elementary transformation to some previously created one—like, for instance, the identity transformation created via `schema_transformation/2`.

In the general case, the combination and creation of transformations is attained via predicates of the form:

$$\langle name \rangle(\texttt{+Pipeline}_{in}\texttt{:}\ \ \textit{ref}\texttt{, +}A_1\texttt{, ..., +}A_n\texttt{, -Pipeline}_{out}\texttt{:}\ \ \textit{ref}\texttt{)}$$

where $\langle name \rangle$ is the name of the transformation being appended to $\texttt{Pipeline}_{in}$, while $A_1, \ldots, A_n$ are transformation-specific parameters, and $\texttt{Pipeline}_{out}$ is the output parameter to which the newly created transformation is bound.

The ML-Lib currently supports 3 predicates of this sort, and further ones may be defined following the same syntactical convention. These are `normalize/3`, `one_hot_encoding/3`, and `attributes_delete/3`, and their details are described later in this paragraph. Here we focus on the overall design which is aimed at supporting the declaration of *pipelines* of transformations, via conjunctions of goals:

```
theory_to_schema(OriginalSchema),
schema_transformation(OriginalSchema, T₀),
transformation₁(T₀, arg₁, T₁),
    ⋮
transformationₘ(Tₘ₋₁, argₘ, Tₘ),
schema_transformation(FinalSchema, Tₘ)
```

Following this convention, logic programmers may declaratively construct the pipeline of transformations to be applied to `OriginalSchema` to produce `FinalSchema`, in such a way that each variable $\texttt{T}_i$, for $i \in \{0, \ldots, m\}$ is bound to an object summarising all transformation steps from 0 to i.

Normalization. A dataset's columns can be normalised in such a way that, for each column, the mean is 0 and the standard deviation is 1. Such kind of transformations may alter the dataset while leaving its schema unaffected. A normalization transformation can be created via the following predicate:

```
normalize(+Pipeline_in: ref, +Attributes: list|atom,
                      -Pipeline_out: ref)
```

There, parameter `Attributes` must be bound to either a list of attribute names or indexes – denoting the columns to be normalized –, or the '`all`' atom—denoting a situation where all columns should be normalized.

One Hot Encoding. A dataset's target attributes whose type are categorical with k-admissible values can be replaced by k binary attributes, via one-hot encoding (OHE) transformations. Such kind of transformations alter both the dataset and its schema. A OHE transformation can be created via the following predicate:

```
one_hot_encode(+Pipeline_in: ref, +Attributes: list|atom,
                      -Pipeline_out: ref)
```

There, parameter `Attributes` must be bound to a list of attribute names or indexes denoting the columns to be one-hot encoded.

Attributes Deletion. Columns may be dropped from a dataset and its schema via attribute deletion transformations. Such kind of transformations alter both the dataset and its schema. An attribute deletion transformation can be created via the following predicate:

```
one_hot_encode(+Pipeline_in: ref, +Attributes: list|atom,
                      -Pipeline_out: ref)
```

There, parameter `Attributes` must be bound to a list of attribute names or indexes denoting the columns to be dropped.

Fitting transformations to data. To support aim 3, the ML-Lib provides the following predicate:

```
fit(+Transformation_in: ref, +Dataset: ref,
          -Transformation_out: ref)
```

which works by tuning `Transformation`$_{in}$ over `Dataset`, producing a new transformation, whose reference is unified with `Transformation`$_{out}$.

The new transformation may be identical to the input one, in case the latter does not require tuning—such as in the case of OHE. Conversely, in case it does need tuning – as in the case of normalization –, the output transformation may actually be different than the original one. Fitting a composite transformation of course has the effect of fitting all its components, recursively.

Applying transformations to data. Finally, to support aim 4, the ML-Lib provides the following *bi-directional* predicate:

```
transform(?Data_in: ref|compound, +Transformation: ref,
                ?Data_out: ref|compound)
```

which can either apply a transformation or its inverse depending on either entire datasets or their rows, depending on how arguments are passed.

In particular, `Data`$_{in}$ and `Data`$_{out}$ can be either dataset references, or compound terms, denoting single rows. Of course, applying a (possibly inverse) transformation to a row (resp. entire dataset) shall produce a row (resp. entire dataset) in return.

The predicate applies `Transformation` to `Data`$_{in}$ in case the latter parameter is instantiated, unifying the transformed result with `Data`$_{out}$. Conversely, it applies the inverse of `Transformation` to `Data`$_{out}$ in case the `Data`$_{in}$ parameter is uninstantiated while the former is not. When this is the case, the transformed result is unified with `Data`$_{in}$.

12.2.4 Predictors

Predictors are stateful entities which can be *trained* over a dataset to later draw *predictions* on new data matching the same schema. In the general case, all predictors may require a number of *hyper parameters* to be specified upon creation, and a number or *learning parameters* to be provided upon training. Both kinds of parameters aim at regulating the predictor behaviour, either in general or during training, and their actual values must be decided by the user.

Given the large number of possible predictors from the data science literature, the ML-Lib just fixes the syntactical convention to support predictors creation, other than the API to support both training and drawing predictions. Notably, as for transformations, the ML-Lib assumes predictors to be represented in object form, and therefore manipulated via reference terms.

Creating predictors. The ML-Lib constrains predictor-creating predicates to comply to the following syntactical convention:

$$\langle name\rangle\texttt{(+}H_1\texttt{, ..., +}H_n\texttt{, -Predictor: } \textit{ref}\texttt{)}$$

where $\langle name\rangle$ is the name of the predictor type being instantiated, while $H_1, \ldots, H_n$ are predictor-type-specific hyper-parameters, and `Predictor` is the output parameter to which the newly created predictor is bound.

The ML-Lib currently supports one predicate of this sort – namely, the `neural_network/2` predicate, described later in this section –, yet further ones may be defined following the same syntactical convention.

Training. Regardless of their nature, predictors can be trained on data via the following predicate:

$$\texttt{train(+Predictor}_{in}\texttt{: } \textit{ref}\texttt{, +Dataset: } \textit{ref}\texttt{, +Params: } \textit{list}\texttt{,}$$
$$\texttt{-Predictor}_{out}\texttt{: } \textit{ref}\texttt{)}$$

The predicate accepts $\texttt{Predictor}_{in}$ as the predictor to be trained, the `Dataset` it should be trained upon, and a list of predictor-specific `Params`. Behind the scenes, the predicate exploits a predictor-specific learning algorithm to train $\texttt{Predictor}_{in}$, possibly following the suggestions/constraints carried by `Params`. Once the training has been completed, a reference to the trained predictor is bound to $\texttt{Predictor}_{out}$, and the execution of the predicate succeeds.

Learning Parameters. The `Params` argument of `train/4` must be instantiated with a list of learning parameters aimed at controlling and constraining the execution of a learning algorithm. In the general case, each parameter is a term of the form:

$$\langle name\rangle\texttt{(}\langle value\rangle\texttt{)}$$

where $\langle name\rangle$ is a functor describing the purpose of the parameter, while $\langle value\rangle$ is an arbitrary term acting as value for the parameter.

In the particular case of neural networks, the ML-Lib admits the following learning parameters

- `max_epochs(N: `*`integer`*`)` limiting the amount of epochs[2] to be performed while training a NN;
- `batch_size(N: `*`integer`*`)` defining the amount of training samples to be taken into account in each single step of the learning algorithm;

[2]i.e., the amount of times the learning algorithm works through the entire training dataset

- `learning_rate(R: real)` defining the step size in a gradient descent learning process;
- `loss(Function: atom)` dictating which loss function should be optimised during training (admissible values include: `mse` for mean squared error, `mae` for mean absolute error, `cross_entropy`, etc.)

Other sorts of learning parameters may be added to the ML-Lib, targeting both NN or other sorts of predictors.

Drawing predictions. Regardless of their nature, *trained* predictors can be exploited to draw predictions from data – e.g. from a whole dataset or a single row –, via the following predicate:

```
predict(+Predictor: ref, +InputData: ref|compound, -Prediction:
                    ref|compound)
```

The predicate accepts a `Predictor` (which must have been previously trained via `train/4`), and some `InputData` – which may either be reference to a dataset object, or a compound term denoting a single row –, and uses the `Predictor` to compute a prediction for each data entry in `InputData`. Predictions may consist of either a single row or a whole dataset, depending on how many data entries are contained in `InputData`. In both cases, the `Prediction` output parameter is unified with the predicted row/dataset.

In case `InputData` is bound to a full dataset including one or more target columns, those target columns are ignored while computing predictions. Conversely, when `InputData` is bound to a list of values, the ML-Lib considers them all as input values.

Classification. As many predictors – there including NN – are technically tailored on *regression* tasks (where predicted values are real numbers), it is a common practice for data scientists to map *classification* tasks (where predicted values are categorical) onto regression tasks, to make it possible to address them via regressors. The mapping commonly works as follows. A classification task requiring input data to be classified according to $k \in \mathbb{N}_{\geq 0}$ classes, can be conceived as a regression aimed at predicting continuos vectors $\mathbf{y} \in \mathbb{R}^k$ from the same input data. Given a particular input datum $\mathbf{x}$, and the corresponding prediction $\mathbf{y}$, the i^{th} component of $\mathbf{y}$ – namely, y_i – could then be interpreted as the confidence of $\mathbf{x}$ being classified as an example of the i^{th} class. Depending on the nature of the classification task at hand, the confidence values in $\mathbf{y}$ could be jointly interpreted following several strategies. In a situation where classes are mutually exclusive, one may use function $argmax_i(y_i)$ to select the most likely class of $\mathbf{x}$. Otherwise, if classes can overlap, one choose a confidence threshold θ and classify **x** according to all those classes i such that $y_i \geq \theta$.

The ML-Lib supports classification out of regressors via the following predicate:

```
classify(+Prediction: ref|compound, +Strategy: compound,
     +Classes: list, -Classification: ref|compound)
```

which accepts a `Prediction` computed via `predict/3` – be it a single row or a whole dataset –, a classification `Strategy`, a list of `Classes`, and an output parameter, `Classification`, which is bound to a container for as many categorical predictions as in `Prediction`.

Notably, while the `Classes` parameter must consist of a list of (at least 2) class names, admissible values for the `Strategy` parameter are determined by the `classification/1` predicate, defined as follows:

```
classification(argmax).
classification(threshold(Th)) :- numeric(Th).
```

meaning that currently the ML-Lib only supports classification via the `argmax` or threshold-based strategies—despite further strategies may be added following the same syntactical notation.

Assessing Predictions. Predictors can be assessed by comparing their *actual* predictions with a test dataset containing *expected* predictions, having no overlap with the data used during training. Several scoring functions can be used to serve this purpose, like, for instance mean squared/absolute error (MSE/MAE) or R^2 for regressors, as well as accuracy, recall, or F1-Score for classifiers.

The ML-Lib supports assessing a predictor via a number of predicates following the same syntactical convention:

```
⟨name⟩(+Actual: ref|list, +Expected: ref|list, -Score: real)
```

where ⟨*name*⟩ is the name of the scoring function of choice, `Actual` is either a dataset or a list containing the actual predictions produced by the predictor under assessment, `Actual` is either a dataset or a list containing the test data, and `Score` is the output parameter to be unified with the score value computed whenever the predicate is executed.

Notable cases of scoring functions are, for instance: `mse/3`, `mae/3`, `r2/3`, `accuracy/3`, `recall/3`, or `f1_score/3`, while further ones may be added following the same syntactical convention.

Neural Networks

Neural networks are a particular sort of predictor. They consist of directed acyclic graphs (a.k.a. DAG) where vertices are elementary computational units called neurons, and edges (a.k.a. synapses) are weighted.

Topologically, neural networks are organised in layers, and data scientists design them by specifying *(i)* how many layers compose the network, *(ii)* how many neurons compose each layer, *(iii)* which activation function is used by each layer – and therefore by each neuron therein contained –, and *(iv)* how are layers – and therefore their neurons – interconnected with their predecessors and successors in the DAG. Hence, a NN's hyper-parameters should provide information about such aspects.

The ML-Lib provides the following predicate to construct NN-like predictors:

```
neural_network(+Topology: ref, -Predictor: ref)
```

There, `Topology` is a reference to an object describing the overall architecture of the network, and, in particular its layers.

Layers. Layered architectures are commonly composed by at least one input layer – whose neurons simply mirror the input data –, and one output layer—whose neurons' output values jointly represent the NN prediction. In the between an arbitrary amount of layers of different sorts may be defined—e.g. dense, convolutional, pooling, etc. In all such cases, declaring a layer implies specifying its sort, size (in terms of neurons), and activation function.

The ML-Lib supports the declaration of layered architectures similarly to how it supports pipelines of transformations. There are two major sorts of predicates to serve this purpose:

```
input_layer(+Size: integer, -Layer: ref).
⟨type⟩_layer(+Previous: ref, +Size: integer, +Activation: ref,
             -Layer: ref).
```

The former predicate, `input_layer/2`, aims at creating a `Layer` of a given `Size`. The size should match the amount of input attributes in the training dataset. This is the entry point of any cascade of predicates aimed at creating a layered architecture.

Conversely, the latter predicate pattern, ⟨*type*⟩`_layer/4` is matched by a number of actual predicates aimed at creating intermediate or output layers. There ⟨*type*⟩ denotes the type of the layer. Regardless of their type, these predicates accept a reference to some `Previous` layer, whose output synapses are connected to the layer under construction, in a way which depends by its type. They also accept the `Size` of the layer to be constructed, and the `Activation` function its neurons should employ. Finally, they all accept an output parameter, `Layer`, to which a reference to the newly created layer is bound, in case creation succeeds.

The `dense_layer/4` predicate is a notable case matching the aforementioned pattern. It aims at declaring a layer whose neurons are *densely* connected with its predecessor's ones—in the sense that, each neuron of the predecessor has an outgoing synapsis towards each neuron of the dense layer. Layers of such a sort are commonly exploited as intermediate. Conversely, layers declared via the `output_layer/4` predicate – again matching the aforementioned pattern – are commonly *final* in any well formed NN architecture.

So, for instance, an ordinary multi-layered perceptron (MLP) composed by 1 input layer with 4 neurons, 1 hidden layer with 7 neurons, and 1 output layer with 3 neurons, where all neurons exploit the sigmoid activation function, can be declared as follows:

```
input_layer(4, I),
dense_layer(I, 7, sigmoid, H),
output_layer(H, 3, sigmoid, O),
neural_network(O, NN)
```

There variable `I` is bound to the input layer, variable `H` is bound to the hidden layer, and `O` is bound to the output layer, whereas `NN` is bound to a MLP predictor whose architecture comprehends `I`, `H`, and `O`.

Activation Functions. The behaviour of neurons should be finely tuned via their activation function. Indeed, all layer-creating predicates of the form ⟨*type*⟩`_layer/4` expect an activation function to be provided by the user. Admissible activation functions are regulated by the `activation/1` predicate, defined below:

`activation(`***identity***`).`	denoting $f(x) = x$
`activation(`***sigmoid***`).`	denoting $f(x) = 1/(1 + e^{-x})$
`activation(`***tanh***`).`	denoting $f(x) = tanh(x)$
`activation(`***relu***`).`	denoting $f(x) = max(0, x)$

while others may be possibly added.

12.3 Technology-related aspects

The ML-Lib requires blending several programming paradigms and languages. Indeed, it is intended to be used by logic programmers and via logic programs – e.g. Prolog programs –, yet it explicitly requires some underlying OO library supporting

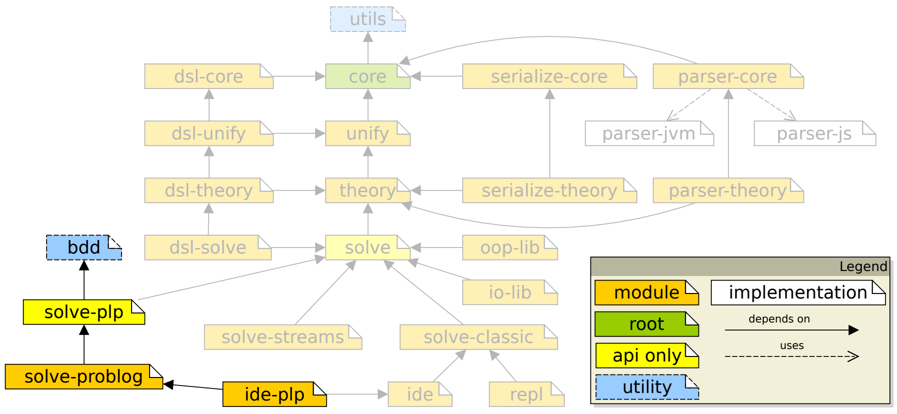


Figure 12.5: Localization of the ML-Lib into the 2P-Kt ecosystem

ML facilities. Hence, from a technical perspective, it requires at least *(i)* a LP technology supporting the definition of custom predicates, possibly triggering the execution of OO code *(ii)* an OO technology supporting the declaration, training, and exploitation of ML predictors.

In this section we discuss the choice of 2P-Kt as the underlying LP technology. However, at the time of writing, this introduces a technological constraint on the JVM platform. Hence, here we also report the technological selection process aimed at selecting the best OO technology supporting ML on the JVM. Spoiler alert: we select DeepLearning4J (DL4J) because of its superior maturity/usability trade-off.

12.3.1 2P-Kt as the underlying logic ecosystem

2P-Kt [CCO21a] is a logic ecosystem rebooting the tuProlog [DOR01] project and providing a number of loosely coupled logic facilities, including but not limited to knowledge representation via terms and clauses, unification, and logic resolution. In particular, 2P-Kt provides such facilities via a Kotlin-based object-oriented API, which eases the construction of applications where LP and OOP are blended together. As a Kotlin library, 2P-Kt can currently run on the JS, JVM e Android platforms.

As shown in fig. 12.5, the ML-Lib is realised as yet another module extending the 2P-Kt ecosystem. In particular, it builds upon *(i)* the knowledge representation facilities offered by the *:core* module, *(ii)* the logic unification facilities offered by the *:unify* module, *(iii)* the clause in-memory indexing facilities offered by the *:theory* module, *(iv)* the general-purpose API for logic resolution offered by the *:solve* module, *(v)* the IO predicates offered by the *:io-lib* module, *(vi)* the OOP↔LP interoperability facilities offered by the *:oop-lib* module, other than, of course *(vii)* DL4J.

With respect to LP solutions, 2P-Kt provides two particular facilities which are key enabler for the ML-Lib. The first one is the notion of *generator*, introduced in [CCO21b] and briefly summarised in section 12.3.1, while the second one is the notion of *reference* term, described in section 12.3.1.

Generators for LP–OOP interoperability

Briefly speaking, generators are gateways mediating the invocation of OO code from logic programs. Thanks to generators, ordinary logic predicates can be implemented in OOP, in such a way that, whenever a goal $p(X_1, \ldots, X_n)$ is met as part of some resolution process, some corresponding OO function (i.e. the generator) is called. The function may then exploit any OO facility available in the local runtime—there including any ML-related library. Eventually, the function must *lazily* return a (possibly infinite) stream of data, as well as a single result. Each returned datum is interpreted back in logic as a solution for the goal p/n which triggered the generator in the first place. Each solution may then carry possible assignments for the variables $X_1, \ldots, X_n$, which may then be taken into account in the remainder of the resolution process.

So, essentially, generators allow for bi-directional communication among a logic solver and the underlying OO runtime. Information may be propagates from the logic realm to the OO one as ordinary predicate arguments, instantiated at invocation time. Information may be back-propagated from the OO realm to the logic one as variable assignments, instantiated at return time.

Generators are supported in 2P-Kt via the `:solve` module. Notably, they are the basic mechanism supporting the manipulation of schemas, datasets, transformations, and predictors in the ML-Lib.

Reference Terms and the `:oop-lib`

2P-Kt's `:core` module provides knowledge representation facilities via terms and clauses. This is achieved via a hierarchy of types – namely, the "term hierarchy", depicted in fig. 12.6 –, supporting the representation of structured or clausal information, within the logic realm.

However, these abstractions alone are not sufficient to support LP–OOP interoperability, as they only support representation of knowledge in logic form. In other words, ordinary logic-based knowledge representation can only represent terms (i.e. alphanumeric or numeric constants, data structures, or variables) or Horn clauses. Objects and types from the OOP realm cannot be represented.

To overcome these issues, 2P-Kt comes with a `:oop-lib` module extending the term hierarchy as shown in fig. 12.6. Briefly speaking, the extension consists of

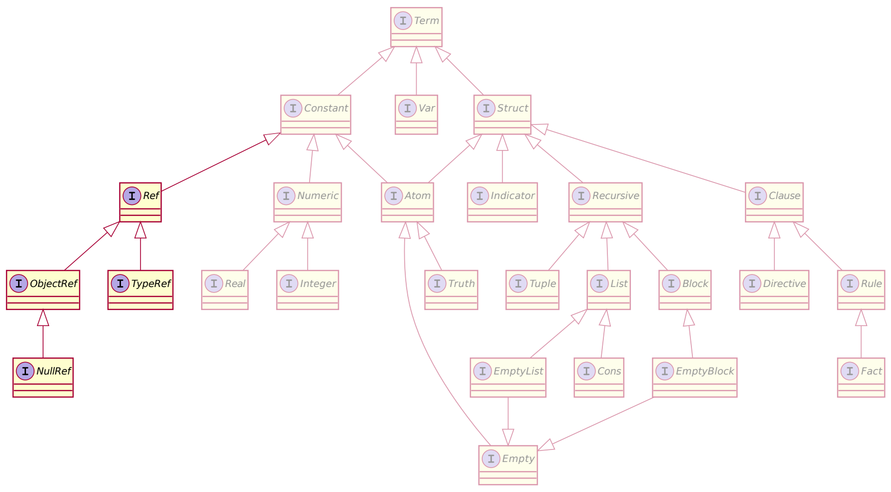


Figure 12.6: Reference terms and their localization within the 2P-KT term hierarchy

a new sort of constant term, namely *references*—formally denoted by the `Ref` type in the diagram. Reference terms can either reference *objects* (`ObjectRef`) or types (`Types`)—as OOP languages may support both instance and shared (a.k.a. static) methods. The *null* reference (`NullRef`) is a particular type of object reference denoting the lack of reference.

Thanks to reference terms, objects can be represented in the logic realm, and carried around as part of the resolution process. Combined with the generators feature, this feature enables logic solvers to take objects into account during resolution. As constant terms, references can be bound to variables, unified, and passed as arguments to predicates, in both verses. Whenever an object must be created or manipulated, a logic solver may delegate the task to the OOP realm, via some *ad-hoc* generator.

Notably, this mechanism as well is fundamental to support the manipulation of schemas, datasets, transformations, and predictors in the ML-Lib.

12.3.2 Selecting the underlying OO library

Here we present a technological analysis aimed at selecting the most adequate library to support our ML-Lib. The analysis is subject to a unique – yet very strong – constraint, derived from the nature of 2P-KT: JVM compatibility.

Notably, 2P-KT offers unique features which can hardly be found in other LP ecosystems. These include the possibility *(i)* to exploit LP facilities *as a library* in OOP, *(ii)* to define custom sorts of terms, and *(iii)* to implement custom logic predicates via OOP. Unfortunately, however, these possibilities come at a price. Any piece of OOP code willing to interoperate with 2P-KT must be compliant with some of the platforms 2P-Kt supports—which currently means the JVM, JS, and Android. Of these the JS platform is considered poorly adequate for ML – as it is a platform mostly suited for Web-related solution –, while Android is too narrow—as targeting Android alone would imply the ML-Lib to only be usable on smart devices. Ad-hoc native bindings towards C/C++ based solution for ML could be considered, but they would involve high costs in terms of engineering and maintenance—which would be prohibitive for and early research project.

Hence, in the reminder of this subsection we focus on JVM-based solutions for ML, supporting all the sorts of entities described in the previous sections—namely, Schemas, Datasets, Transformations, Predictors, and, in particular, Neural Networks.

DeepLearning4J (DL4J). DL4J [Tea] is a Java-based solution for neural networks and deep learning, inspired to Python-based projects such as Tensorflow and Pytorch.

The project is currently developed and maintained by the Konduit team[3], and supported by the Eclipse Foundation[4]. The project comes with a wide gamma of functionalities covering the many needs of the deep learning practitioner. These include ND4J, i.e. an efficient library for tensors manipulation and scientific computations; Datavec, i.e. a library for loading data from various sources into tensors; Samediff, i.e. a Tensorflow/Pytorch-like differential engine for the execution and optimization of complex computational graphs; and Arbiter, i.e. a tool for hyper-parameters optimization.

In particular, the Samediff engine is explicitly designed to rely on ND4J, thus guaranteeing good performance in spite of low memory requirements. Overall, DL4J is interoperable with a number of relevant technologies from the ML playground, including but not limited to Spark (for the execution of distributed learning pipelines), CUDA (for hardware accelerating tensor manipulations), Tensorflow (as it supports the '`.h5`' data format), as well as the Open Neural Network Exchange (ONNX) format. Furthermore, DL4J is shipped with a number of pre-trained models (model zoo), and many code snippets exemplifying their usage.

The project is actively maintained and rapidly evolving. It currently includes 56 contributors and more than 12.3k stars on GitHub. Its codebase is well documented on the project homepage, and several tutorials are provided as well. Finally, DL4J is an open source project made publicly available under the terms of the Apache 2.0 License.

[3]`https://konduit.ai`
[4]`https://www.eclipse.org`

Smile. Smile [Smi21] is a Java-based solution for machine learning, inspired the Python-based project SciKit-learn. It started as a side-project of dr. Haifeng Li., in 2014. Nowadays, it is amongst the widest (in terms of features) independent frameworks for ML targeting the JVM platform. Notably, it does not simply support neural networks training and exploitation, but the whole gamma of ML predictors as well—including, but not limited to SVM, decision trees, linear models, random forests, etc. In particular, the Smile library includes pure-Java types and methods for linear algebra and tensor manipulation, dataset manipulation and visualization, and plenty of algorithms for both supervised and unsupervised ML, there including statistical methods, classification and regression, and a number of meta-heuristics.

Unfortunately, NN support is very limited in Smile, as it only supports multi-layered percepts with dense layers, trained on the CPU—i.e. with no hardware acceleration. Automatic differentiation support is lacking, as well as the possibility to construct custom network architectures.

The project is actively maintained, mostly by dr. Li himself, and other 54 contributors, and it currently involves 5.4k stars on GitHub. Its codebase is well documented on the project homepage, and several tutorials are provided as well. Finally, Smile is an open source project made publicly available under the terms of the Apache 2.0 License.

Neuroph. Neuroph [STCG$^+$21] is a Java framework for neural networks developed by at University of Belgrade through a number of PhD and master students' theses. The project aims at providing a lightweight platform for exploiting and visualising neural networks, mostly in research and for teaching.

Similarly to other technologies presented so far, Neuroph supports the definition, training, assessment, and visualization of neural networks. However, the amount of supported features is quite restricted, automatic differentiation is lacking, and the support for creating custom NN architectures is limited. These limitations are coherent with the design goal of creating a didactic environment for neural networks newbies, yet they may result prohibitive for complex projects needing to be competitive w.r.t. the state of the art.

The project is currently being maintained by Zoran Sevarac and other 21 contributors, and it currently involves 51 stars on GitHub. Finally, Neuroph is an open source project made publicly available under the terms of the Apache 2.0 License, since version 2.4, or the LGPL license before that version.

Tensorflow for Java (TF4J). Tensorflow [AAB$^+$15] is a popular framework for ML, and in particular deep learning developed by the Google Brain team. It consists of a differential engine written in C++ and involving several bindings towards high-level languages, such as Python.

TF4J is the official binding of Tensorflow for the JVM. Unfortunately, however, TF4J does not currently have the same maturity of the Python binding, nor it covers the same gamma of features. Notably, TF4J does not support the support the whole API described by the official Tensorflow documentation. In particular it does not support all the features introduced in Tensorflow 2.0, such as the Keras API—supporting the manipulation of NN in terms of layers.

The project involves more than 450 stars and 64 contributors on GitHub. It consists of an open source project made publicly available under the terms of the Apache 2.0 License.

Deep Java Library (DJL). DJL [Ama21] is a Java framework for deep learning application developed by Amazon. It supports the definition, training, and exploitation of neural network, via one or more pluggable differential engines The currently available differential engines are the Pytorch, MXNet, and Tensorflow ones. For all such engines, a number of pre-trained neural architectures (model zoos) are provided as well. Hence, DJL essentially consists of a common API for training or loading, and running NN targetting state of the art differential engines.

Despite DJL is currently actively maintained, it is to be considered in embryonic stage. At the time of writing, it mostly focuses on image processing and computer vision applications, while it does not include functionalities for handling datasets and their schemas.

The project currently involves 48 contributors and 2.4k stars on GitHub. It consists of an open source project made publicly available under the terms of the Apache 2.0 License.

H2O. H2O [H2O16] is a ML, deep learning, and distributed computing platform for the JVM, developed by H2O.ai. It consists of a JVM-based service running on top of a cluster of machines, possibly involving Spark or Hadoop. Clients may perform ML or data processing via a ReSTful API exposed by that service. Implementations for that API are available for the R, Python, and JS languages. Clients may request data processing or ML operations on data, to the service, which then exploits the joint computational power of the cluster to perform those operations, following the map-reduce approach [DG08]. Hence, H2O is a nice solution for Big Data scenarios.

The project is actively maintained and it involves 154 contributors and 5.7k stars on GitHub. It consists of an open source project made publicly available under the terms of the Apache 2.0 License.

Project	**Developer / Funder**	**Maturity**	**Development Status**	**Documentation**
DL4J	Konduit/Eclipse Fundation	Beta	Actively Developed	Excellent
Smile	Haifeng Li	Stable	Developed	Excellent
Neuroph	Zorac Severac	Stable	Maintained	Acceptable
TF4J	Google Brain Team	WIP	Developed	Excellent
DJL	Amazon Web Services	Alpha	Actively Developed	Good
H20	H20.ai	Stable	Developed	Good
Weka	Waikato University	Stable	Developed	Excellent

Table 12.1: Recap of the analysed technologies and their features

Project	**Neural networks**	**Linear Algebra**	**Dataset pre-processing**	**Other sorts of predictors**	**Differential engine**
DL4J	Yes	Yes	Yes	No	Yes
Smile	No	Yes	Yes	Yes	No
Neuroph	Yes	No	No	No	No
TF4J	Yes	No	No	No	Yes
DJL	Yes	Yes	No	No	Yes
H20	Yes	Yes	Yes	Yes	Unknown
Weka	Yes	Yes	Yes	Yes	via DL4J

Table 12.2: Recap of the analysed technologies and the functionalities they support

Weka. Weka [WFH11] is a platform for ML, and, in particular data mining, developed in Java by the Waikato University. It includes a wide collection of algorithms supporting not only supervised and unsupervised learning, but also data pre-processing, exploration, and visualization. Weka can be exploited as a Java library or as a graphical application for data mining. In the latter case, no coding skills are required for using Weka.

Concerning NN, Weka provides basic support for multi-layer perceptrons only, without relying on automatic optimization. Hence, the construction of arbitrarily-structured NN is not supported directly, but rather via the WekaDeepLearning4J extension[5]—which wraps DL4J behind the scenes.

Weka is a well-established and widely-adopted solution within the data mining community. The official Web page includes and excellent documentation, including tutorials and videos. It consists of an open source project made publicly available under the terms of the GPL 3.0 License.

Comparison, discussion, and selection

Here we motivate the choice of DL4J as the underlying OO library supporting our first implementation of the ML-Lib.

The selection criterion leading our choice takes into account a number of aspects summarised in tables 12.1 and 12.2. In particular, for each technology, we analyse some features aimed at assessing its usability, stability, and durability (from a software engineering perspective), other than which and how many functionalities it offers.

Functionalities are, of course, of primary importance. For instance, we prioritize NN support via a differential engine, as that would enable great flexibility in the construction, training, and exploitation of NN of different shapes. Functionalities supporting the loading and manipulation of datasets and their schemas are equally relevant. Conversely, supporting the full gamma of ML predictors is a nice to have (but not strictly required) functionality.

Business oriented features are, however, of paramount importance as well. Even in presence of all required functionalities, technologies involving a large development team, a big funding organization, a large user base, or a high-paced development/release history should be preferred—as they are less likely to be dismissed or become unmaintained in the future.

Along this line, DL4J is the most adequate option. Indeed, the project provides a sufficiently large functionality support, and it comes with a good documentation and encouraging expectations for what concerns its maintenance and future development—as it is backed by the Eclipse Foundation, and it has a large user base. Furthermore, it comes with a well engineered object-oriented API, whose meta-model is close the one we adopted while designing the ML-Lib. Smile covers a wide gamma of functionalities, but it is poorly suited for NN, and, in particular, it is an independent project with a small development team and a low-paced release rate. Neuroph is mainly suited for educational application, and it provides too simple API. Furthermore, while it is currently maintained, it is not actively developed any more, and no major releases are expected in the future. TF4J is still too embryonic and its API is too limited, at the moment, to serve the purpose of the ML-Lib. DJL is a promising project – both from the technical and business-oriented perspectives –, however it is currently better suited for use cases where NN trained elsewhere needs to be executed on the JVM. Furthermore, it provides less functionalities than DL4J, while serving similar purposes. H2O is an interesting solution, but its client-server architecture may greatly complicate the low level design and implementation of our ML-Lib, hence it is not considered adequate for bootstrapping a prototype. Finally, Weka another perfect solution

[5]`https://deeplearning.cms.waikato.ac.nz`

for backing our ML-Lib: it supports the full gamma of required functionalities, it has a large contributors list, it is well established, and it has a large user base. The main reason why we prefer DL4J over Weka is that the latter fully supports NN only through the former. So, when prototyping our ML-Lib we choose to rely on simplest possible setting – namely, vanilla DL4J –, instead of an integrated solution involving Weka + WekaDeepLearning4J—which may however be pursued in the future.

12.4 ML-Lib Examples

Here we exemplify the usage of the ML-Lib to serve the purposes described in section 12.1.1.

From a LP perspective, our examples assume the existence of a logic solver/language exploiting some implementation of the ML-Lib. For the sake of simplicity, we assume a Prolog solver is employed. Hence, examples consists of Prolog scripts, possibly involving standard Prolog predicates.

From a ML perspective, our examples assume a very simple scenario where a neural-network classifier is trained on the well known Iris dataset[6]. The resulting NN is then exploited to write a simple hybrid predicate aimed at classifying unseen Iris instances.

Declarative ML. Declarativity is a key benefit of our symbolic approach to ML. The ML-Lib supports declarative ML in several ways, as exemplified by listings 12.1, 12.2, 12.3, and 12.5.

In particular, listing 12.1 shows how the schema and data entries of the Iris dataset can be treated in logic. Notably, the Iris data set contains 150 rows describing as many individuals of the Iris flower. For each exemplary, 4 continuous input attributes – *petal* and *sepal width* and *length* – are recorded, other than a categorical target attribute—denoting the actual Iris *species*. There are three particular species of Iris in this data set – namely, Setosa, Virginica, and Versicolor –, and the 150 examples are evenly distributed among them—i.e., there are 50 instances for each class. The Prolog script describes the Iris dataset's schema in clausal form, as discussed in section 12.2.1. It also declares two predicates – namely, `iris_schema/1` and `iris_dataset/1` – aimed at letting the logic programmer retrieve either the schema or its dataset in object form. More precisely, `iris_schema/1` attempts to read the schema from the local theory, while `iris_dataset/1` attempts the load the dataset from a CSV file. Listing 12.4 (presented later in this section) reports a similar scenario where the dataset as well is

[6]`https://archive.ics.uci.edu/ml/datasets/iris`

Listing 12.1: Dataset loading from file

```
% schema declaration
attribute(0, sepal_length, real).
attribute(1, sepal_width, real).
attribute(2, petal_length, real).
attribute(3, petal_width, real).
attribute(4, species, categorical([setosa, versicolor, virginica])).
schema_target([species]).
schema_name(iris).

% reading schema from theory
iris_schema(S) :- theory_to_schema(S).

% dataset loading
iris_dataset(D) :- read_dataset('/path/to/iris.csv', csv, D).
```

Listing 12.2: Pre-processing pipeline

```
%  declaring & fitting the preprocessing pipeline
preprocessing_pipeline(Dataset, Schema, Pipeline) :-
    schema_transformation(Schema, Step0),
    Classes = [petal_width, petal_length, sepal_width, sepal_length],
    normalize(Step0, Classes, Step1),
    one_hot_encode(Step1, [species], Step2),
    fit(Step2, Dataset, Pipeline).
```

Listing 12.3: Neural network structure declaration

```
% neural network declaration
multi_layer_perceptron(Nin, Nhidden, Nout, NN) :-
    input_layer(Nin, IL),
    hidden_layer(IL, Nhidden, H),
    output_layer(H, Nout, softmax, O),
    neural_network(O, NN).

hidden_layer(L, [], L).
hidden_layer(L, [N | M], H) :-
    dense_layer(L, N, relu, L1),
    hidden_layer(L1, M, H).
```

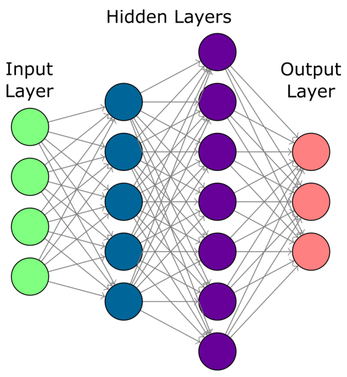


Figure 12.7: Simplified representation of a multi-layer perceptron having 4 input neurons, 2 hidden layers with 5 and 7 neurons respectively, and 3 output neurons

loaded from the local theory.

Listing 12.2 exemplifies the declaration of a pre-processing pipeline aimed at normalising the input attributes of any `Dataset` having the same `Schema` of Iris, other than one-hot encoding its output attributes. The resulting `Pipeline` is then fitted against the provided `Dataset`, and bound to the corresponding output argument.

In turn, Listing 12.3 presents a general purpose predicate aimed at defining multi-layered perceptron predictors with an arbitrary amount of hidden layers. This is made possible by the `multi_layer_perceptron/4` predicate, which requires the caller to provide the amount of neurons to be instantiated for *(i)* the input layer (`Nin`), *(ii)* the output layer (`Nout`), and *(iii)* for each hidden layer (`Nhidden`). Notably, `Nhidden` should consist of a list in integers, denoting the amount of neurons for each hidden layer – from the outermost to the innermost –, while the total amount of integers corresponds to the amount of hidden layers. The resulting neural network predictor is then bound to the `NN` output argument. So, for instance, a NN such as the one depicted in fig. 12.7 can be declared as follows:

```
multi_layer_perceptron(4, [5, 7], 3, NN)
```

Finally, listing 12.5 declares an end-to-end ML workflow aimed at selecting and training the best NN architecture to tackle Iris classification. Further details about that listing are discussed later in this section. For the moment, we simply stress the declarative nature of the script which can be regarded as a formal – yet human-readable – specification of a classifier training workflow.

Symbolic data sources. In an hybrid system integrating both symbolic reasoning and sub-symbolic learning, it may be useful to perform ML upon data expressed in logic form. This requires logic theories to act as symbolic data sources.

Listing 12.4: Dataset loading from the local theory

```
% schema declaration
attribute(0, sepal_length, real).
attribute(1, sepal_width, real).
attribute(2, petal_length, real).
attribute(3, petal_width, real).
attribute(4, species, categorical([setosa, versicolor, virginica])).
schema_target([species]).
schema_name(iris).

% dataset definition
iris(5.1, 3.2, 1.4, 0.2, setosa).
iris(4.9, 3, 1.7, 1.2, versicolor).
iris(5.9, 3.4, 1.1, 0.9, virginica).
/*
 *    ... other entries here...
 */

% reading schema from theory
iris_schema(S) :- theory_to_schema(S).

% reading dataset from theory
iris_dataset(D) :-
    iris_schema(S),
    theory_to_dataset(S, D).
```

Our ML-Lib makes it possible to support such scenario, as exemplified in listing 12.4. The script is assumed to replace listing 12.4 in those situation where the Iris dataset is logically described in clausal form. Here, the `iris_dataset/1` attempts to load the data from the local theory instead of a file.

Model selection via resolution. The automatic exploration of a search space subtended by logic resolution could be exploited to perform model selection. Indeed, model selection essentially consists of an exploration of the hyper and learning parameters space, looking for the best possible values—i.e. those hyper and learning parameters assignments corresponding to well-performing predictors on the available training set.

Accordingly, the ML-Lib supports expressing and performing model selection in logic, as exemplified in listing 12.5. There hyper, learning, and workflow parameters are expressed as logic facts, and the `params/2` predicate is defined to enumerate all possible combinations of theirs—e.g. via Prolog's backtracking mechanism. The `model_selection/5` predicate is in charge of stepping through all such parameters with the purpose of selecting, and training all corresponding NN predictors which attain a sufficiently high predictive performance—denoted by the `target_performance/1` fact. For each trained predictor, the predicate outputs not only a reference to the `Predictor` itself, but also its `Performance`, and the affine `Transformation` to be applied to each datum for which predictions should

Listing 12.5: Declarative description of a ML workflow aimed at selecting the best hyper and learning parameters for a NN classifier. Ancillary predicates invoked in this snippet are reported in listing 12.6

```
/* Hyper paramenters */
hidden_layers([10]). hidden_layers([20, 10]). hidden_layers([30, 20, 10]).

/* Learning paramenters */
max_epochs(30). max_epochs(50).
batch_size(32). batch_size(16).
learning_rate(0.01). learning_rate(0.1).
loss(cross_entropy).

/* Workflow paramenters */
target_performance(0.90).
test_percentage(0.2).

/* Generates all possible hyper and learning params combinations */
params(
    [hidden_layers(H)],
    [iterations(X), learning_rate(Y), batch_size(Z), loss(L)]
) :- hidden_layers(H),
     max_epochs(X),
     learning_rate(Y),
     batch_size(Z),
     loss(L).

/* Generates and trains all possible Predictors for the given Dataset and Schema,
    */
/* whose Performance is at least target_performance. */
model_selection(Dataset, Schema, Predictor, Transformation, Performance) :-
     test_percentage(R), target_performance(T),
     random_split(Dataset, R, TraingSet, TestSet),
     preprocessing_pipeline(TraingSet, Schema, Transformation),
     transform(TraingSet, Transformation, ProcessedTrainingSet),
     params(HyperParams, LearnParams),
     train_cv(TraingSet, HyperParams, LearnParams, P),
     P >= T,
     multi_layer_perceptron(4, HyperParams, NN),
     train(NN, TrainingSet, LearnParams, Predictor),
     transform(TraingSet, Transformation, ProcessedTestSet),
     test(NN, TestSet, Performance).

/* Example of training query: */
?- iris_dataset(D), iris_schema(S), model_selection(D, S, P, _, A).
```

Listing 12.6: Ancillary predicates used in listing 12.5. Each predicate denotes one particular step of a model selection workflow

```prolog
/* Trains a NN multiple times, over Dataset, using the provided Params. */
/* Returns the AveragePerformance over a 10-fold CV. */
train_cv(Dataset, HyperParams, LearnParams, AveragePerformance) :-
    findall(
        Performance,
        train_cv_fold(Dataset, 10, HyperParams, LearnParams, Performance),
        AllPerformances
    ),
    mean(AllPerformances, AveragePerformance).

/* Trains a NN once, for the k-th round of CV. */
/* Returns the Performance over the k-th validation set. */
train_cv_fold(Dataset, K, HyperParams, LearnParams, Performance) :-
    fold(Dataset, K, Train, Validation),
    train_validate(Train, Validation, HyperParams, LearnParams, Performance).

/* Tranis a NN on the provided TrainingSet, using the provided Params, */
/* and computes its Performance over the provided ValidationSet. */
train_validate(TrainingSet, ValidationSet, HyperParams, LearnParams, Performance)
    :-
    multi_layer_perceptron(4, HyperParams, 3, NN),
    train(NN, TrainingSet, LearnParams, TrainedNN),
    test(NN, ValidationSet, Performance).

% Computes the Performance of the provided NN against the provided ValidationSet
test(NN, ValidationSet, Performance) :-
    predict(NN, ValidationSet, ActualPredictions),
    accuracy(ActualPredictions, ValidationSet, Performance).
```

be drawn using that predictor.

More precisely, the predicate `model_selection/5` works by

1. splitting the provided `Dataset` into a `TrainingSet` and a `TestSet`, according to a split ratio (`R`) declared by the `test_percentage/1` fact
2. declaring and fitting a pre-processing `Transformation` aimed at normalising the `TrainingSet`'s input attributes, and one-hot encoding its output attributes
3. applying such `Transformation` to the `TrainingSet`, hence producing a `ProcessedTrainingSet`
4. stepping through all possible hyper (`HyperParams`) and learning (`LearnParams`) parameters combinations,
5. training each corresponding predictor, via 10-fold cross validation (CV), and computing its average validation-test performance (`P`)
6. skipping each hyper and learning parameters combination such that the average performance `P` is lower than the target performance `T`
7. re-training a full-fledged MLP on the whole `TrainingSet`, for each parameters combination such that `P >= T`
8. testing that MLP against the `ProcessedTestSet` – attained by applying the aforementioned `Transformation` to the `TestSet` as well –, thus computing the MLP actual `Performance`

In other words, the `model_selection/5` represents a declarative, and pretty general, workflow for model selection—which may be adapted to other supervised learning tasks with minimal changes. It relies on a number of ancillary predicates declaring some particular steps of the workflow, and exemplifying many ML-Lib functionalities. These are reported in listing 12.6. For instance, `train_cv/4` is in charge of performing 10-fold CV on a given `Dataset`, to assess a given `HyperParams`–`LearnParams` combination, to then compute the `AveragePerformance` of the 10 predictors constructed in this way. Each single fold of a `K`-fold CV process is managed by the `train_cv_fold/5` predicate, which in turn exploits `train_validate/5` predicate to train and validate each single predictor. Finally, the `test/3` predicate can be exploited to either test or validate a predictor depending on whether the test or validation set is provided as argument.

Under these hypotheses, a model selection workflow for the Iris dataset may be triggered via a concise logic query such as:

Listing 12.7: Exploitation of the NN classifier trained in listing 12.5 to create an hybrid predicate – namely `iris/5` – aimed at classifying Iris flowers

```
/* assumption: */
:- iris_dataset(D), iris_schema(S), model_selection(D, S, N, T, _), !, assert(
    iris_nn(N, T)).

/* hybrid iris classifier */
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, Species) :-
   X = [SepalLength, SepalWidth, PetalLength, PetalWidth],
   iris_nn(Network, Transformation),
   transform(X , Transformation, ActualX),
   predict(Network, X, Y),
   classify(Y, argmax, [setosa, versicolor, virginica], Species).
```

```
?- iris_dataset(D), iris_schema(S), model_selection(D, S, P, _, A).
```

If all aspects of the model selection workflow are correctly declared, the query should provide multiple successful solutions corresponding to all trained predictors (`P`) and their test-set accuracies (`A`).

Hybrid reasoning. Finally, listing 12.7 shows the exploitation of a trained NN predictor as a predicate aimed at classifying (possibly) unseen instances of the Iris flower. The script serves a twofold purpose: it exemplifies the ML-Lib functionalities aimed at drawing predictions out of trained ML predictors, and, in particular, it provides an example of an *hybrid* reasoner—where symbolic and sub-symbolic AI seamlessly interoperate.

The script assumes a fact of the form `iris_nn(N, T)` is available into the solver's KB, storing a reference to a trained NN predictor (`N`) and to the affine transformation (`T`) to be applied to each datum the predictor should be fed with. Such assumption may be satisfied, in Prolog, by a query such as the following one:

```
?- iris_dataset(D), iris_schema(S), model_selection(D, S, N, T, _),
                        !, assert(iris_nn(N, T)).
```

which selects and trains a single NN and stores it into the solver's dynamic KB.

Under such assumption, logic programmers may write an `iris/5` predicate such as the one shown in listing 12.7. The predicate allows the caller to classify Iris instances by triggering a previously trained NN, and by letting it draw predictions on the data row attained by composing the predicate's arguments—via the `predict/3` predicate. The prediction is then converted into a class constant – via the `classify/4` predicate –, which is in turn bound to the output parameter of `iris/5`—namely `Species`.

It is worth to be highlighted that, from the caller perspective, the `iris/5` described so far is undistinguishable from a purely symbolic predicate serving the

Listing 12.8: A purely symbolic classifier for Iris flowers, functionally equivalent to the hybrid one from listing 12.7

```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalWidth =< 0.78.
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalLength >= 2.86, PetalLength < 4.91.
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica).
```

same purpose (i.e., Iris classification) and having the same name and arity—such as the one described in listing 12.8.

12.5 Recap and Research Perspectives

In this chapter, we propose a logic API supporting the seamless integration of logic solvers with sub-symbolic AI, and, in particular neural-network-based supervised ML.

Stemming from a domain analysis aimed at identifying the major computational entities involved in a supervised ML workflow, we design our API in terms of computational entities and the operations/functionalities they should support. We then reify our API into a set of logic predicates composing the ML-Lib—i.e., an abstract logic library which any goal-oriented solver may support, there including Prolog ones. Both the syntax and the semantics of each single predicate are discussed, as part of the major contribution of this chapter. Architectural and technological requirements are discussed as well.

Among the most relevant requirements, we stress the need of realising the ML-Lib as a façade towards some lower-level OOP library for ML. Furthermore, to support the prototyping of our ML-Lib on top of the 2P-Kt logic ecosystem – which technologically unifies the LP and OOP realms –, we also require the underlying OOP library for ML to be JVM-compliant. Hence, we elaborate on a technological discussion aimed at selecting the most adequate JVM technology for ML.

Finally, we provide a number of usage examples aimed at showing the potential of the ML-Lib. In particular, we discuss examples where our logic API supports *declarative* ML (possibly from symbolic data sources), model selection via resolution, and hybrid reasoning. Indeed, the ML-Lib enables the user to formally define ML workflows in a way which is both human- and machine-interpretable, focussing on what should be done, rather than how.

Hybrid reasoning, in particular, is the most relevant contribution of ours. It consists in the seamless integration of logic and sub-symbolic AI at the functional level. In fact, thanks to our ML-Lib, trained sub-symbolic predictors may be used in LP as ordinary predicates.

Research Perspectives. In the future, we expect contributions to stem from our ML-Lib along two different research threads. The first thread concerns the exploitation of the ML-Lib to create hybrid systems, where LP and ML are integrated in manifold ways. This is made possible by our logic API for ML, which reduces the abstraction gap among LP and ML, as well as the ML-Lib, which lowers the technological barriers preventing the integration of symbolic and sub-symbolic AI. The second thread concerns the extensions of the ML-Lib, which should be eventually delivered to cover currently unsupported functionalities—as well as other ML predictors than NN.

Chapter 13

Bridging LP and XAI

This chapter contains contributions from the following works of ours: [SCCO21]

Artificial neural networks (ANN), support vector machines (SVM), and other data-driven predictors are nowadays among the most-used tools to face a wide range of different tasks involving machines learning (ML) from data [RPM12]. In all those cases, the learning activity consists of tuning the parameters of predefined algorithms in order to maximise their predictive capability w.r.t. the data at hand.

The major drawback of state-of-the-art ML algorithms is that they are inherently *opaque*, meaning that they do not provide any intelligible representation of what they learn from data. This is why most of those algorithms are considered as black boxes (BB) which only represent knowledge in a *sub-symbolic* way. Nevertheless, despite their sub-symbolic operation may prevent human users from understanding *how* they work, BB – and, in particular, ANN – are being increasingly applied to support forecasting and decision making in many different fields – including, but not limited to, marketing, customer/user profiling, social networks, predictive maintenance, etc. – because of their unprecedented predictive performance.

There exist, however, critical applications where black-box predictions or recommendations are unacceptable: for instance, healthcare, finance and law domains, or any other area of knowledge where decision making may affect critical aspects of human lives—e.g. health, wealth, freedom, etc. In all those cases, it is of paramount importance to rely on *explainable* predictions, recommendations, or suggestions, in order to let humans retain accountability and liability over the decision or choices they make.

As further discussed in section 6.3, in this thesis we commit to symbolic knowledge extraction (SKE) as the preferred means to derive *post-hoc* explanations for sub-symbolic predictors. However, despite the wide adoption of SKE, a unified and general-purpose software technology supporting it is currently lacking. In other words, the burden of implementing SKE algorithms is currently on data scientists alone, who are likely to realise custom solutions on a per-need basis. Other than producing inertia w.r.t. the adoption of SKE in modern data, such a lack of viable technologies is somewhat anachronistic in the data-driven AI era, where a plethora of libraries and frameworks are flourishing, targeting all major programming paradigms and platforms, and making state-of-the-art machine learning algorithms easily accessible to the general public—cf. SciKit-Learn[1] for Python, or Smile[2] for the Java Virtual Machine (JVM).

Accordingly, in this chapter we present the design of PSyKE, a general-purpose Platform for Symbolic Knowledge Extraction aimed at filling the gap between the current state of the art of SKE and the available technology. More precisely, PSyKE is conceived as an open library where different sorts of knowledge extraction algorithms can be implemented, exploited, or compared. PSyKE supports rule extraction from both classifiers and regressors, and makes the extraction procedure as transparent as possible w.r.t. the underlying BB, depending on the particular extraction procedure at hand. Notably, it also supports the extraction of first-order logic (FOL) clauses, with the twofold advantage of providing human- and machine-interpretable rules as output. Rules can then be used as either an explanation for the original BB, or as a starting point for further symbolic computations. PSyKE is designed as a general framework that can be specialised on multiple runtimes and programming environments. The technology currently comes along with two main implementations to enable SKE in two key domains: *(i)* the JVM implementation for the symbolic AI domain, and *(ii)* the Python implementation for the sub-symbolic AI domain. SKE is one of the elements that act as a bridge between the two domains, as well as the production of logic programs [Llo90, MN96] as output—in particular expressed as Prolog programs [CR93].

A number of experiments involving rule extraction on both classification and regression tasks – performed on well-known public data sets – are discussed to demonstrate the versatility of PSyKE. To this end, we run framework experiments against various BB predictors and perform a comparison between different extraction procedures applied to the same task. The comparison takes into account the fidelity of the extracted rules (w.r.t. the original BB) and predictive performance w.r.t. the data.

Accordingly, the remainder of this chapter is organised as follows. Section 13.1 describes the state of the art for SKE as well as some background notion to fully understand the work. Section 13.2 presents the design of PSyKE, while in section 13.3 some use cases showing how PSyKE can be exploited are reported. Conclusions are drawn in section 13.4.

[1]`https://scikit-learn.org/stable` [Last accessed April 17, 2022]

[2]`https://github.com/haifengl/smile` [Last accessed April 17, 2022]

13.1 State of the Art

In this section we firstly overview the state of the art for symbolic knowledge extraction (section 13.1.1). Then, we delve into the details of a selection of extraction algorithms—namely, the ones PSyKE implementation currently supports (section 13.1.1–section 13.1.1). The algorithm selection is performed by keeping *variety* (rather than exhaustivity) in mind, so as to demonstrate the operation and versatility of PSyKE. In particular, our aim is to exemplify the many application scenarios that a data scientist may meet—e.g., extraction from either classifiers or regressors, trained on either categorical or continuous data.

Finally, we briefly outline the currently-available software object-oriented frameworks for ML (section 13.1.2). The overview is meant to make the chapter self-contained, given that one of these frameworks provides PSyKE with pure ML functionalities—in particular, the design of PSyKE assumes basic classification or regression support to be available as a software library.

13.1.1 Knowledge Extraction

According to [CCSO20], a computational system is considered *interpretable* if human beings can easily understand its operation and outcomes. The majority of modern ML predictors, however, sacrifice interpretability to enhance the predictive performance, thus becoming increasingly complex. They do so by merely focusing on learning highly-predictive – yet *sub-symbolic* – input-output relations from data, while neglecting any attempt to make such relations *symbolic*, i.e., intelligible for human. For this reason, ML algorithms are often called *black boxes* [Lip18].

To mitigate interpretability issues without sacrificing predictive performance, a number of authors from the XAI community have proposed means to produce *ex-post* explanations for sub-symbolic predictors—most notably, ANN and SVM. Explanations, in this case, consist of *surrogate* predictors trained to mimic the ones to be explained, as closely as possible.

In practice, among the manifold proposals, some authors describe methods to extract if-then-else rules [CS94, HBV06, SCO21], whereas others propose methods extracting decision trees [CS95]. While the shape of the extracted knowledge may vary from an extraction procedure to another, all the proposed methods share the trait of extracting *symbolic* (i.e. human-intelligible) knowledge out of *sub-symbolic* ML predictors. Given a trained predictor and a knowledge-extraction procedure applicable to it, the extracted rules/trees act as *explanations* for that predictor – or as a basis to build some –, provided that they retain high *fidelity* w.r.t. the underlying predictor [CCSO20]. The extracted knowledge may then enable further manipulations, such as merging the *know-how* of two or more BB models [CCOC19].

According to [CCO20], knowledge extraction methods can be categorised along three orthogonal dimensions, namely: *(i)* the sort of learning tasks they support, *(ii)* the shape of the symbolic knowledge they produce, *(iii)* their *translucency*—i.e., the sort of BB algorithms they can extract symbols from.

About dimension *(i)*, one can distinguish among algorithms targeting classification tasks, regression tasks, or both. In other words, some extraction algorithms can only deal with BB classifiers – e.g. Rule-extraction-as-learning [CS94] (REAL, henceforth), TREPAN [CS95] and others [BD08, MBVGV07] –, while others can only deal with BB regressors – such as ITER [HBV06], GridEx [SCO21], REFANN [SLZ02], ANN-DT [SAG99] and RN2 [SN02] –, and only a few can handle both—such as G-REX [KJN08] and CART [BFOS84]. Notably, virtually all extraction methods proposed so far are tailored on *supervised* machine learning. To the best of our knowledge, no rule extraction procedure has been proposed targetting unsupervised or reinforcement learning tasks.

As far as dimension *(ii)* is concerned, decision rules [Fre14, HDM$^+$11, MP91] and trees [Qui93, Qui87] are the most widespread human-understandable shapes for extracted knowledge, thus most methods produce one of these two structures. In both cases, decision rules or nodes are expressed in terms of the same input/output data types the original BB has been trained upon. So, for instance, an extraction procedure processing a BB classifier for N-dimensional numerical data, over K classes, will likely output rules/trees involving one or more *predicates* over N input variables $x_1, \ldots, x_n$ and K possible outcomes. In any case, however, extraction algorithms are further categorised w.r.t. the particular sort of predicates their output rules/trees may contain. Accordingly, conjunctions/disjunctions of inequality (e.g. $x_i \gtrless c$), or interval inclusion/exclusion expressions (e.g. $x_i \in [l, u]$) are commonly exploited for numerical data, while equality (e.g. $x_i = c$) or set-inclusion $x_i \in \{c_1, c_2, \ldots\}$ expressions may be exploited for categorical data. Finally, *M-of-N* rules are yet another possible choice in the case of boolean data.

The translucency dimension [ADT95] from item *(iii)* refers to the need/capability of the extraction procedure to "look into" the *internal* structure of the underlying BB—i.e., to what extent it has to be taken into account during the extraction procedure. There are two major ways for categorising knowledge extractors w.r.t. translucency. During the extraction process, *decompositional* extractors may take into account the internal structure of the BB they operate upon, while *pedagogical* ones do not. For this reason, pedagogical approaches are usually more general – despite potentially less precise –, thus they can be applied to every BB predictor regardless of its kind, structure, and complexity.

The quality of knowledge-extraction procedures is evaluated through different indicators depending on the task to solve, for instance, fidelity and predictive

Extraction Algorithm	Task	Translucency	Required Features	Knowledge Shape	Exhaustive
REAL	Classification	Pedagogical	Binary	Rule list	No
Trepan	Classification	Pedagogical	Binary	Decision tree	Yes
Iter	Regression	Pedagogical	Continuous	Rule list	No
GridEx	Regression	Pedagogical	Continuous	Rule list	Yes
Cart	Classification and regression	Pedagogical	Continuous or binary	Decision tree	Yes

Table 13.1: Summary of the knowledge-extraction algorithms supported by PSyKE

performance measurements [TS93]. In particular, the former indicates how well the extracted knowledge mimics the underlying black-box predictions, whereas the latter measures the explanator predictive power w.r.t. the data. In all cases, measurements should be taken via the same scoring function used for assessing the BB performance—which in turn depends on the task it performs. In the particular case of black-box classifiers, examples of performance measurements are accuracy, precision, recall, and F1-score; for BB regressors, the mean absolute/squared error (MAE/MSE) and the R^2 scores could be exploited.

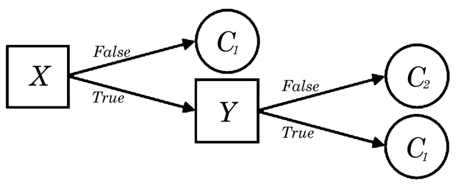
In the following, we provide a more detailed description of some extraction procedures currently supported in the PSyKE framework, grouped by the task they address—i.e. classification, regression, or both of them. All the algorithms are *pedagogical*, thus they only rely on the BB inputs and outputs, and do not inspect the inner structure of the underlying BB. This is why they can be applied to any kind of arbitrarily-complex BB. In any case, structured data are required. The same algorithms are summarised in table 13.1.

Extraction from Classifiers

Rule-extraction-as-learning. REAL [CS94] is a pedagogical algorithm to extract conjunctive rules from trained BB classifiers by using a learning process driven by sampling and queries. Output rules can be either *if-then* or *M-of-N* rules. An example of *if-then* output rule is the following: *Output class is C if* $\{X, Y, Z\}$ *are Trueand* $\{U, V\}$ *are False*, where *U, V, X, Y, Z* are one-hot encoded input features. Such features are *True* if they are 1, *False* otherwise.

Output rules are disjunctive normal form expressions; each term is the conjunction of a data set feature subset, adequately generalised by dropping each non-discriminant antecedent. REAL cannot handle real-valued features, but only binary ones.

Trepan. Trepan [CS95] is a pedagogical algorithm able to extract symbolic and comprehensible model representations from trained classifier BB by inducing a decision tree approximating the BB represented concept. It usually maintains high



$$\text{Output class is } C_1 \text{ if} \begin{cases} (X \text{ is } \textit{True}) \text{ or} \\ (X \text{ is } \textit{False} \text{ and } Y \text{ is } \textit{True}) \end{cases}$$

Output class is C_2 otherwise.

Figure 13.1: Example of TREPAN output tree (left) and corresponding rules (right)

fidelity levels w.r.t. the underlying BB while being comprehensible and accurate. It is general in its applicability and well scalable with complex models or problems. However, as REAL, it cannot be applied to real-valued features.

In fig. 13.1 an example of output tree is reported. Internal nodes are represented by squares, while leaves are circles. Variables reported inside the internal nodes are the split criteria for the subtree creation. C_i are the class labels corresponding to each leaf.

Extraction from Regressors

Iter. ITER [HBV06] is a pedagogical algorithm for building predictive rules from trained BB regressors of any kind. Its main idea is to iteratively expand a number of hypercubes until they cover the whole input space. Each of them is finally converted into an *if-then* rule of the following format: *Output constant is C if* $X_1 \in [l_1, u_1]$ *and ... and* $X_k \in [l_k, u_k]$, where l_i and u_i are the lower-bound and the upper-bound for variable X_i. There, the preconditions of the rule describe a k-dimensional hypercube. Indeed, ITER supports continuous input features, differently from the other algorithms presented so far.

GridEx. GridEx [SCO21] is another pedagogical extraction algorithm for regressors; it is an extension of ITER aimed at overcoming its major drawback: non-exhaustivity. GridEx adopts a top-down approach to iteratively partition the input feature space in a user-defined number of hypercubes or in an automatic way accordingly to a user-defined strategy based on feature importance. As ITER, GridEx produces *if-then* rules and only accepts data sets with real-valued input features, but it is always exhaustive by design. Since the procedure associates each hypercube to a rule, a merging phase is performed after every iteration as an optimisation to reduce the number of rules.

General-Purpose Extractors

Cart. CART [BFOS84] is an algorithm for building decision trees that can be used to face both classification and regression tasks. CART is not properly a

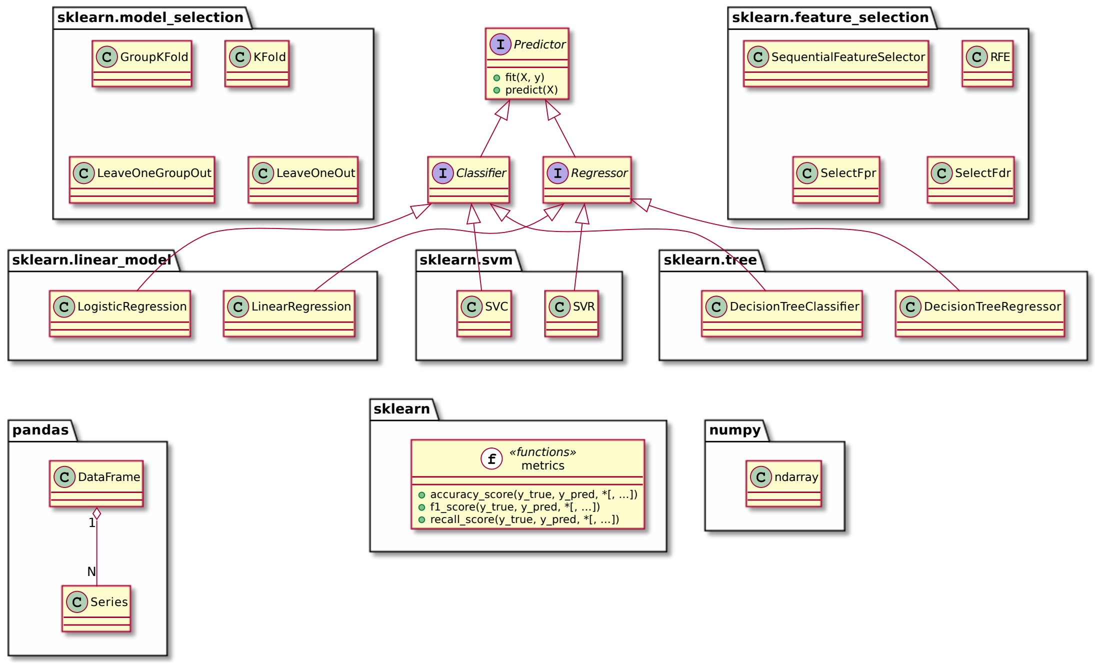


Figure 13.2: Overview on (a small portion of) the API of the Scikit-Learn library supporting classification and regression tasks

knowledge-extraction procedure, but from its output tree it is straightforward to obtain a rule tree, since each node of the CART tree corresponds to a constraint on a certain feature and thus each path from the root to a leaf is a single complete classification or regression rule. The algorithm can be summarised via the following instructions: *(i)* initialise the tree root node; *(ii)* find optimal splits and add new internal nodes and leaves accordingly; *(iii)* stop the algorithm based on one or more criteria—e.g., leaf number or tree depth. Pruning algorithms can be applied to reduce the number of leaves.

13.1.2 Object-Oriented Programming Frameworks for ML

In order to make ML solutions easily available, a number of frameworks – especially exploiting object-oriented programming (OOP) – have been developed. Such frameworks usually provide users with powerful abstractions for modelling BB as well as for performing data set pre-processing, feature engineering and predictive performance measurements. Among the most supported ML models, there are ANN, SVM, and decision trees for both classification and regression tasks. The most complete frameworks also provide utility packages to ease the data set reading from (and writing into) files and to perform feature selection, beyond many other tools for natural language processing, linear algebra, and data visualisation.

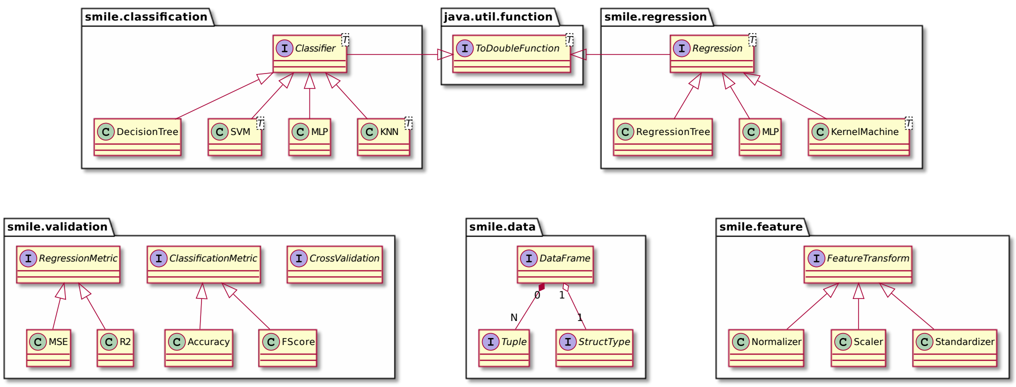


Figure 13.3: Overview on the API of the Smile library supporting classification and regression tasks

Examples of state-of-the-art OOP frameworks for ML are Scikit-Learn [PVG+11] for Python and Smile [Smi21] for the JVM.

At the conceptual level, the design of such frameworks is similar, and similar are the set of functionalities they provide. For this reason, the design of PSyKE assumes an underlying ML framework to be available, from which major ML-related facilities can be borrowed. In the general case, these facilities include:

F1 a base type for supervised predictors (i.e., either classifiers or regressors), providing a common interface for their training and their exploitation for inference;

F2 a base type for classifiers and a base type for regressors, as particular subsorts of predictor, providing a way to discern among the two at runtime;

F3 *ad-hoc* sub-types for well-known predictors – such as ANN, SVM, decision trees, etc. –, encapsulating the algorithmic strategies for their training and their inference, and supporting the manipulation of individual predictors as instances of those types;

F4 a type for representing data sets and their schemas, providing a common interface for their manipulation (scaling/translating columns, adding/removing features or instances, renaming/moving/transforming features, etc.);

F5 the possibility to parametrise predictors' training by letting developers specify hyper-parameters;

F6 the possibility to select one or more scoring functions to assess the performance of trained (or in-training) predictors;

F7 support for ML best practices, such as test set separation, cross-validation, grid search, etc.;

F8 support for feature engineering (there including selection, encoding, missing values imputation, etc.);

F9 the possibility to inspect the internals of any individual predictor (e.g. synapses in neural networks, or decision splits for decision trees, etc.), which is necessary to support the implementation of decompositional extraction algorithms.

The abstract architecture of PSyKE can be reified on each programming platform for which a library providing the facilities above exists. Notably, in this chapter, we demonstrate the versatility of our design by describing two different implementations of PSyKE. One implementation is based on Scikit-Learn and targets Python, whereas the other is based on Smile and targets the JVM.

Scikit-Learn

Scikit-Learn [PVG+11] is amongst the most well-known Python libraries for machine learning. It is characterised by a coherent design, an efficient implementation and the wideness of its scope—which covers ML far beyond supervised techniques.

Figure 13.2 depicts a (partial) UML class diagram representing the major interfaces and classes composing Scikit-Learn's supervised learning API. Module names are explicitly indicated to help the user understand the organisation of the library. Notably, Scikit-Learn heavily rely on Python's *duck typing* convention. So, despite no explicit type definition exists for predictors (neither for classifiers nor for regressors) – meaning that facility item **F2** is not supported –, all objects exposing the methods `fit(...)` and `predict(...)` are considered as such—approximating facility item **F1** via convention. Nevertheless, each kind of ML predictor has a dedicated class (facility item **F3**) and each class follows the aforementioned convention. Parametrisation of learning (facility item **F5**) occurs at the instantiation level, via construction parameters.

External packages, such as Pandas[3] and Numpy[4], provide data types for easily managing data sets, features, and tuples (intended as data set columns and rows, respectively). The three Python packages – Scikit-Learn, Pandas, and Numpy – are independently maintained and deployed, despite being perfectly interoperable. In particular, Numpy features efficient implementations for multi-dimensional arrays, while Pandas provides for higher-level abstraction such `DataFrames`, for data sets, or `Series`, for both schemas and instances (facility item **F4**). Scoring functions

[3]`https://pandas.pydata.org` [Last accessed April 17, 2022]
[4]`https://numpy.org` [Last accessed April 17, 2022]

(facility **F6**) are reified as functions having analogous signatures (name apart), hosted by the `sklearn.metrics` module. Conversely, cross-validation, grid search, and other model-selection tools (facility item **F7**) are reified into *ad-hoc* classes as well, hosted by the `sklearn.model_selection` module. Finally, feature extraction and selection (facility item **F8**) are supported via the `smile.feature_extraction` and `smile.feature_selection` modules, respectively, and the many classes therein contained.

Inspectability (facility item **F9**) can be achieved via Python "consenting adults" philosophy, which essentially allows expert programmers to freely access the internals of any Python object—there including Scikit-Learn predictors. Therefore, decompositional extraction algorithms can be implemented on top of Scikit-Learn, provided that the inner functioning of predictors includes enough documentation to let programmers inspect them.

Smile

Smile (Statistical Machine Intelligence and Learning Engine) [Smi21] is defined as "a fast and comprehensive machine learning engine" for Java, and any other JVM-based language—e.g. Scala and Kotlin.

Figure 13.3 depicts a (partial) UML class diagram representing the major interfaces and classes composing Smile supervised learning API. Package names are explicitly indicated to avoid confusion between homonymous classes. Notably, each kind of ML predictor has a dedicated class (facility **F3**) and each class implements either the `Classifier` or the `Regression` interface (facility **F2**). Both interfaces descend from the `ToDoubleFunction` interface, which is, therefore, the most adequate type to represent any supervised ML predictor (facility **F1**). Parametrisation of learning (facility **F5**) occurs at the instantiation level, via construction parameters.

Other packages (not shown in figure) provide data types for easily managing data sets, feature vectors, and tuples (intended as data set columns and rows, respectively) other than all the aforementioned facilities. So, for instance, the `DataFrame`, `StructType`, and `Tuple` types are the basic types for representing data sets, schemas, and instances, respectively (facility **F4**). Scoring functions (facility **F6**) are reified as classes as well, implementing either the `ClassificationMetric` or the `RegressionMetrics` interface. A similar statement holds for cross-validation (facility **F7**). Finally, feature engineering and selection (facility **F8**) are supported via the `smile.feature` package and the many classes therein contained.

Inspectability (facility **F9**) is apparently laying outside the current design goals of Smile: the current API of Smile does not provide any means to observe the internals of trained predictors. Therefore, only pedagogical extraction algorithms can be implemented on top of Smile.

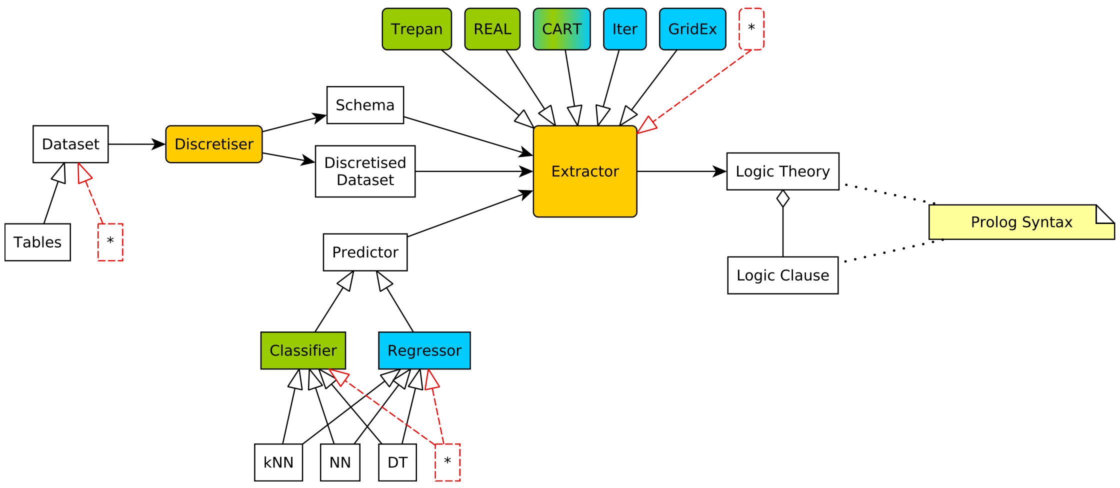


Figure 13.4: PSyKE design

13.2 PSyKE

PSyKE is a software library providing general-purpose support to the extraction of logic rules out of BB predictors by letting users choose the most adequate extraction method for the task and data at hand. PSyKE exposes a unified API covering virtually all extraction algorithms targeting supervised learning tasks. Currently, the implementations of PSyKE involve several interoperable, interchangeable, and comparable extraction procedures – namely, the ones mentioned in section 13.1.1 –, granting access to state-of-the-art knowledge-extraction algorithms to both researchers and data scientists. PSyKE is conceived as an open-ended project, which can be exploited to design and implement new extraction procedures behind a unique API.

Essentially, PSyKE is designed around the notion of *extractor*, whose overall design is depicted in fig. 13.4. Within the scope of PSyKE, an extractor is any algorithm accepting a ML predictor – either a classifier or a regressor – as input, and producing a *theory* of *logic* rules as output.

To perform their job, PSyKE extractors require additional information about the data set the input predictor has been trained upon. In the general case, such information consists of the data set itself and its schema—i.e., a formal description of the names and the data types of all features characterising the data set itself. More precisely, data sets are required to let extraction procedures inspect BB behaviour – and therefore build the corresponding output rules –, whereas schemas are required to let *(i)* the extraction procedure take informed decisions on the basis of the feature *types*, *(ii)* the extracted knowledge be clearer by referring to the feature *names*. For all these reasons, extractors expect a data set and its schema metadata to be provided in input as well.

Many extraction procedures can operate on discrete/binary data only. This is commonly made necessary by the shape of the extracted rules—which consists of simple predicative statements about some feature value. However, it is also very common in data science to meet data sets involving continuous attributes as well. Accordingly, extracting rules out of predictors trained on continuous data may be troublesome in the general case. To circumvent this issue, PSyKE also provides some facilities aimed at discretising (binarising) data sets including continuous (categorical) data. When these are in place, extractors should be provided with the discretised/binarised schema as well, to be able to produce the clearest rules possible.

Accordingly, in the rest of this section we detail *(i)* the general design of the PSyKE library and API, *(ii)* the discretisation facilities, *(iii)* the general shape of the extracted logic theory.

13.2.1 General API

As depicted in fig. 13.5, a pivotal role in the design of PSyKE is played by the `Extractor` interface, defining the general contract of any knowledge-extraction procedure. Technically, each `Extractor` encapsulates a single ML `Predictor` and a particular `Discretization` strategy for the data it operates upon. Under such conditions, an extractor is capable of extracting a `Theory` of logic `Rule`s out of a `DataFrame`, containing the examples the `Predictor` has been trained upon.

A relevant aspect of PSyKE design – and a prerequisite for its understanding –, is that it avoids re-inventing the AI wheel. Thus, PSyKE assumes that some underlying libraries are available on the runtime adopted for implementation, from which AI facilities can be inherited. These include: *(i)* a ML library, exposing *ad-hoc* types aimed at representing data sets, data schemas, or predictors, and *(ii)* a symbolic AI library, exposing *ad-hoc* types for representing and manipulating logic theories, clauses, and rules. From these libraries, PSyKE borrows high-level abstractions, required for its operation, which would be prohibitive to re-design or re-implement from scratch. These include, for instance, the following types:

`DataFrame` — a container of tabular data, where rows commonly denote instances, and columns denote their features, while bulk operations are available to manipulate the table as a whole, as well as any row/column of its;

`Predictor<R>` — a computational entity which can be trained (a.k.a. fitted) against a `DataFrame` and used to draw predictions of type `R`;

`Classifier<R>` — a particular case of predictor where `R` represents a type having a finite amount of admissible values;

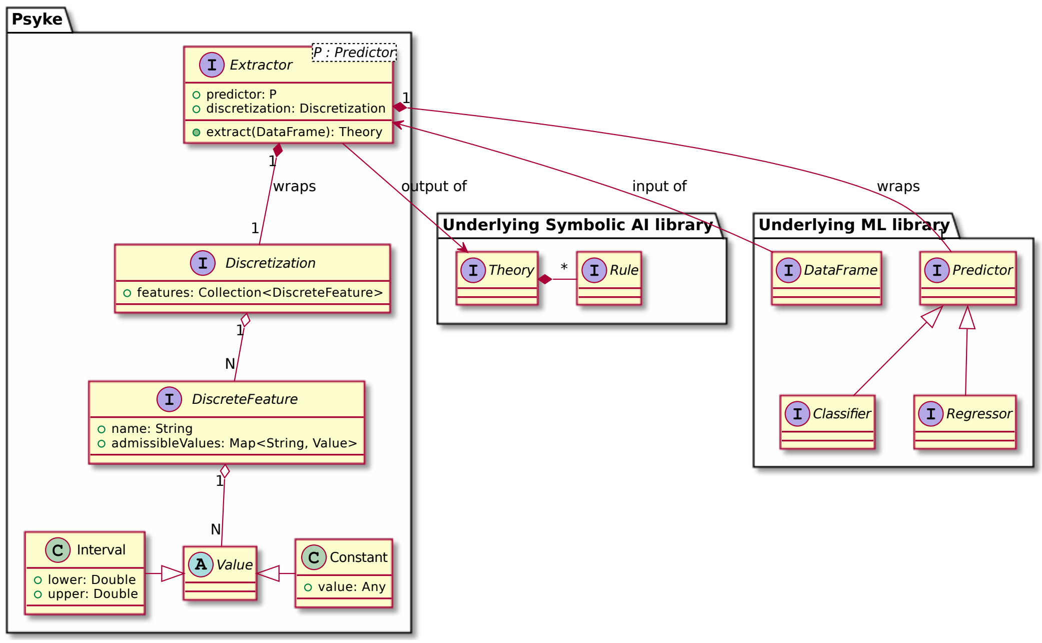


Figure 13.5: PSyKE's `Extractor` interface

`Regressor<R>` — a particular case of predictor where `R` represents a type having a potentially infinite (possibly continuous) amount of admissible values;

`Rule` — a semantic, intelligible representation of the function mapping `Predictor`'s inputs into the corresponding outputs, for a particular portion of the input space;

`Theory` — an ordered collection of rules.

For example, PSyKE may borrow ML-related abstractions – such as `DataFrame`, `Predictor`, or `Classifier` – from either Smile or Scikit-Learn, depending on the particular runtime of choice (among JVM or Python), or any library serving the same purpose and exposing analogous abstractions. Similarly, it may borrow high-level symbolic-AI-related abstractions – such as `Theory` or `Rule` – from 2P-Kt [CCO21a] which supports both the JVM[5] and Python[6] runtimes—and of course any functionally-equivalent library could be used as well.

Upon such premises, PSyKE constructs a notion of `Extractor`—i.e., any method capable of extracting logic `Rule`s out of some trained `Predictor`. In our design, PSyKE extractors are bound to the particular black-box `Predictor` they aim to extract rules from, as well as a `Discretization` strategy for its input space. These fields are provided upon instantiation, and never altered since then. `Extractors` also expose: *(i)* a method for extracting an explainable `Theory` from the `Predictor` – namely, `extract` – and *(ii)* a method to draw predictions by using the extracted rules—namely, `predict`. Of course, prediction implies extraction—meaning that each attempt to use the extracted rules to draw explainable predictions shall trigger extraction first. Both extraction and prediction rely on a `DataFrame` which must be provided by the user upon invocation. It should contain the data the predictor has been trained upon – or some structurally analogous data –, in order to let the extraction algorithm determine the structure of the extracted rules.

Notice that `Predictor`s are parametric types. There, the meta-parameter `R` represents the type of predictions the predictor may produce—unknown at design time. The rules possibly extracted by such predictors – as well as the predictions drawn from them – may vary significantly depending on the particular data and predictors of choice. For instance, when rules must be extracted from mono-dimensional regressors, `R` may be the type of floating point numbers, whereas for multi-class classifiers, `R` may consist of the set of types (like integer, string, etc.). Of course, the rules possibly extracted by such predictors greatly differ, depending on the nature of `R`. However, the proposed API makes it possible to switch between different extraction algorithms and predictors requiring no changes in the architecture of PSyKE, but only minor adjustments in the user code.

[5]`https://github.com/tuProlog/2p-kt`
[6]`https://github.com/tuProlog/2ppy`

13.2.2 Discretisation

A large number of the knowledge-extraction procedures – in the same way as many ML algorithms – require either a discrete or binary input space—i.e. all input features must be either categorical or one-hot encoded, respectively. For instance, REAL and Trepan require exclusively one-hot encoded data, whereas Iter and GridEx require continuous data. Cart can accept both continuous and one-hot encoded features, but not categorical ones. Unfortunately, most real-world applications are described by real-valued variables and measurements, thus making the application of such algorithms impractical. The general way to overcome this limitation is to rely on some discretisation/binarisation method among the many available in the literature—e.g., [DKS95, YWW10, Ker92, HS97, Bou04, KC04, CNVC16].

Briefly speaking, discretisation is the process of transforming a datum from some continuous space $I \subseteq \mathbb{R}$ into a discrete space $\{I_1, \dots, I_n\}$ such that $\forall i, j = 1, \dots, n$: $I_i \subset I \wedge I \equiv \bigcup_i I_i \wedge i \neq j \Leftrightarrow I_i \cap I_j = \varnothing \wedge i < j \Leftrightarrow \forall x \in I_i, \forall y \in I_j : x < y$. Similarly, binarisation (a.k.a. one-hot encoding) is the process of transforming a datum from some discrete space $X = \{x_1, \dots, x_n\}$ into a binary space $B = \{b_1, \dots, b_n\}$ where for each $i = 1, \dots, n$: b_i is 1 if the datum is equal to x_i, 0 otherwise. Of course, these methods imply a considerable increase in the dimensionality of a data set—e.g., one-hot encoding makes categorical attributes with 4 distinct values be converted into 4 different boolean features. This is far from being an issue: in some cases, it is possible to achieve even better classification performances by using discretised attributes rather than continuous, as demonstrated in [EEM21].

PSyKE provides different procedures to manipulate input features: *(i)* a discretisation for continuous features, mapping real intervals into categorical features, and *(ii)* a one-hot encoding for categorical features, mapping exact values to boolean features. Notably, PSyKE traces the input feature transformations by creating a data structure that associates the initial name of the attribute and the newly created features with the corresponding constraints. An example of PSyKE binarisation and corresponding output data structure is reported in fig. 13.6. This example considers the *petal length* attribute of the Iris data set and adopts the default supervised discretisation method available in our framework. The initial continuous feature values are labelled in fig. 13.6 with a), and graphically represented in the b1) plot. PSyKE discretisation algorithm consists in calculating the mean attribute value and the corresponding standard deviation *for each data set class*. An interval is then initialised for each class, with lower and upper bounds equal to the mean value—cf. plot b2). Each interval is iteratively expanded until convergence—i.e., when the whole feature space is covered without overlapping intervals. To achieve this, during every iteration all the intervals are symmetri-

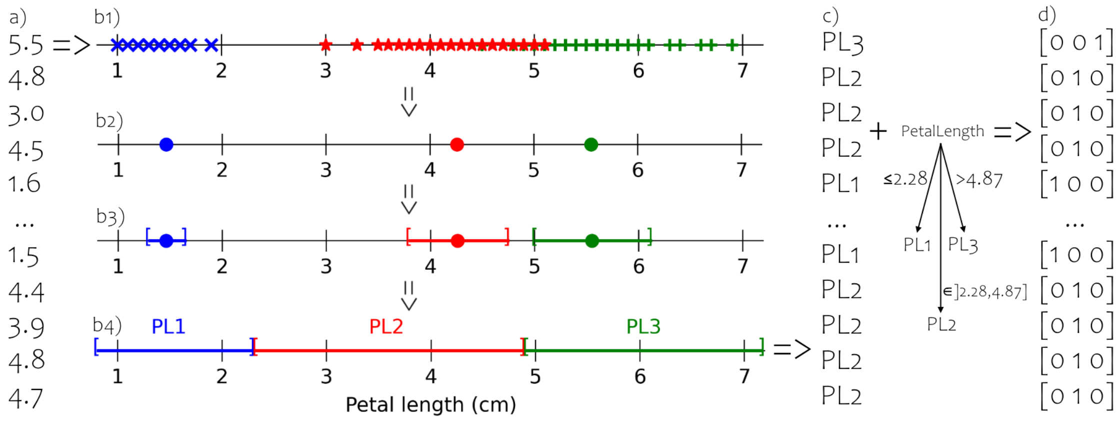


Figure 13.6: PSyKE discretisation and binarisation procedure

cally expanded of a value equal to the corresponding standard deviation – in each direction, as in plot b3) –, in order to create intervals with adaptive size. An expansion is inhibited when *(i)* the lower (upper) bound of an interval exceeds the minimum (maximum) value of the feature space, or *(ii)* two adjacent intervals are overlapping. In the first case, the feature minimum (maximum) value is taken as the final interval lower (upper) bound. In the second case, the overlapping intervals are only expanded up to the mean value between their respective boundaries. When all intervals have been calculated – cf. plot b4) – the continuous attribute values are converted accordingly into categorical values. The discretised output values are labelled in the example with c). In this step PSyKE also produces a data structure for keeping the discretisation details, to be able to produce more compact rules during the extraction procedures. The last step – labelled with d) – is the one-hot encoding of the discrete values into arrays of binary data—i.e., the unique format accepted by several extraction procedures, such as REAL and TREPAN.

13.2.3 Output rules

PSyKE extractors output knowledge in the form of logic theories – i.e., lists of Horn clauses –, notably in Prolog syntax. We choose the Prolog syntax to make them simultaneously interpretable by both humans and machines. More precisely, PSyKE output theories are structured as lists of Prolog rules or facts. Rule heads are $(n+1)$-ary predicates, where n is the number of input features in the data set. These predicates carry n variables – i.e., one for each input feature – and either a constant or a list – i.e., the output value(s) – as argument. Predicate names recall the classification/regression under study. Rule bodies can be empty – if rules are *facts* – or conjunctions of literals where each literal is a predicate expressing inequality, equality, or interval inclusion between attribute actual values and fixed constants calculated through the extraction process.

Accordingly, a rule-extraction procedure targeting a mono-dimensional classification *task* on a data set having n input features and m relevant output values, shall output theories of the following form:

$$\begin{array}{rcl}
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{y}_1) & \texttt{:-} & p_{1,1}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,1}(\bar{\mathtt{X}}).\\
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{y}_2) & \texttt{:-} & p_{1,2}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,2}(\bar{\mathtt{X}}).\\
 & \vdots & \\
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{y}_m) & \texttt{:-} & p_{1,m}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,m}(\bar{\mathtt{X}}).
\end{array}$$

where *(i) task* is the $(n+1)$-ary relation representing the classification or regression task at hand, *(ii)* each $\mathtt{X}_i$ is a logic variable named after the i^{th} input attribute of the currently available data set, *(iii)* $\bar{\mathtt{X}}$ is the n-nuple $\mathtt{X}_1,\ldots,\mathtt{X}_n$, and *(iv)* each $p_{i,j}$ is either a n-ary predicate expressing some constraint about one, two or more variables, or the `true` literal—which can be omitted.

Similarly, a rule-extraction procedure targeting a mono-dimensional regression *task* shall output theories of the following form:

$$\begin{array}{rcl}
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{Y}) & \texttt{:-} & p_{1,1}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,1}(\bar{\mathtt{X}}),\mathtt{Y}\ \texttt{is}\ f_1(\bar{\mathtt{X}}).\\
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{Y}) & \texttt{:-} & p_{1,2}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,2}(\bar{\mathtt{X}}),\mathtt{Y}\ \texttt{is}\ f_2(\bar{\mathtt{X}}).\\
 & \vdots & \\
\langle\mathsf{task}\rangle(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{Y}) & \texttt{:-} & p_{1,m}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,m}(\bar{\mathtt{X}}),\mathtt{Y}\ \texttt{is}\ f_m(\bar{\mathtt{X}}).
\end{array}$$

where $\langle\mathsf{task}\rangle$, $\mathtt{X}_i$, $\bar{\mathtt{X}}$, and $p_{i,j}$ have the same meaning than the classification case, whereas *(i)* f_j is an n-ary function computing the output value for the regression task in the particular portion of the input space handled by the j^{th} rule, and *(ii)* `is/2` is the well-known Prolog predicate aimed at evaluating functions.

Without lack of generality, the aforementioned rule forms assume the case under study to involve *mono-dimensional* classification/regression tasks. In fact, multi-dimensional cases can be tackled by allowing lists of constants – e.g. $\mathtt{[y^{(1)},y^{(2)},\ldots]}$ – in the heads of classifications rules; or lists of variables – e.g. $\mathtt{[Y^{(1)},Y^{(2)},\ldots]}$ – in the head of regression rules, as well as multiple variable assignments in their bodies—e.g. $\mathtt{Y}^{(1)}\ \texttt{is}\ f_j^{(1)}(\bar{\mathtt{X}}),\mathtt{Y}^{(2)}\ \texttt{is}\ f_j^{(2)}(\bar{\mathtt{X}}),\ldots$

The rationale behind our proposed way of structuring rules is straightforward: the j^{th} rule selects a portion of the input space via a number of constraints $p_{1,m}(\bar{\mathtt{X}}),\ \ldots,\ p_{n,m}(\bar{\mathtt{X}})$ – which may be less than n in practice –, and then dictates the most adequate prediction for that portion. In classification tasks, the prediction is constant for the whole portion, whereas in regression tasks it is computed via a function $f_j(\bar{\mathtt{X}})$. This function may itself output a constant in simpler cases, or a local approximation of the predictor's behaviour in the corresponding portion of the input space—e.g. a linear approximator. Notice that, in classification tasks, the total amount of rules (m) may still be greater than the total amount of classes (k), as there may be more than one rule for the same class. In the general case, m is bound to the number of portions of the input space detected by the extraction algorithm of choice.

Currently, the supported sorts of predicates in rules bodies – i.e., the admissible shapes for each $p_{i,j}$ – are as follows:

equality involving a single variable and a constant—e.g. $\mathtt{X} = c$, where c is a constant of any sort (possibly, a number) [7]

inequalities involving a single variable and a constant—e.g. $\mathtt{X} \gtrless c$

interval inclusion involving a single variable and two constants—e.g. `X in` $[l, u]$, where $l, u \in \mathbb{R}$ and $l < u$

interval exclusion like the above, but negated—e.g. `X not_in` $[l, u]$

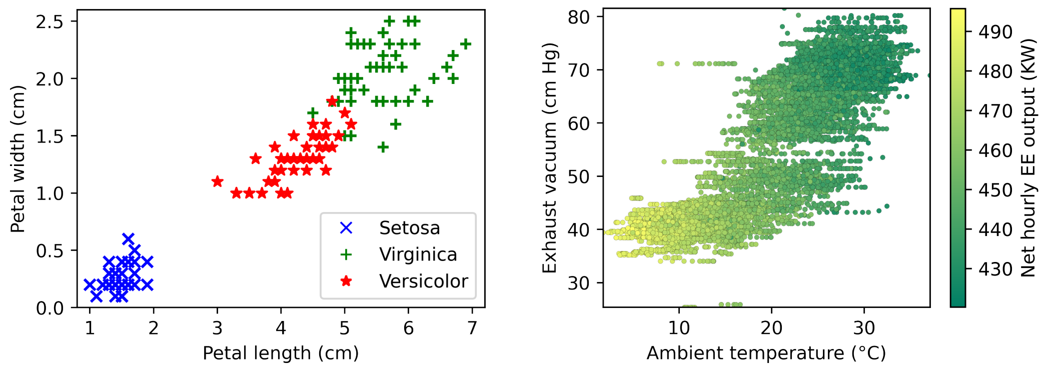
M-of-N involving N variables—e.g. `at_least`$(M, [\mathtt{X}_1, \ldots, \mathtt{X}_N])$, where $M, N \in \mathbb{N}_{\geq 0}$

This holds for both classification and regression tasks.

Despite many other forms can be adopted for the output theories, we argue the proposed one is a good trade-off between human and machine interpretability. In fact, rules of this form are well-formed logic programs, which may be executed by a logic reasoner—such as a Prolog interpreter. Furthermore, the proposed form is open to many sorts of post-processing. For instance, recurrent conjunctions of predicates in rules bodies may be factorised into their own general-purpose rules. Similarly, redundant or cumbersome sub-expressions may be simplified.

However, the proposed form is far from perfection: we plan to explore alternative directions in the future. Noticeably, using Prolog syntax does not impose exploiting also its semantics—i.e., different interpreters can be exploited over such Prolog rules. For instance, on the one side, by exploiting a Prolog interpreter, rules are interpreted as functional (one-way)—meaning that it is possible to compute a prediction given an assignment of all input variables, but it is not possible to generate a correct assignment of those input variables given the expected prediction alone. On the other side, a Constraint Logic Programming solver [GR10, JM94] may interpret the same rules as constraints, and compute coherent assignments for any subset of both input and output variables—providing rules with a relational (two-ways, generative) semantics.

[7]The same result could be attained by allowing constants in rules heads



(a) Iris **(b)** CCPP

Figure 13.7: Sample distribution of the Iris and CCPP data sets. Only the 2 most relevant features are reported.

13.3 Case Study

In this section the effectiveness and versatility of PSyKE are tested by exploiting it in different scenarios—i.e. the Iris data set[8] as classification task and the Combined Cycle Power Plant[9] (CCPP) data set as a regression case study. Examples of output rules extracted with PSyKE algorithms are reported, along with a brief discussion on several possible future improvements for the rule final presentation (see section 13.3.3). Figure 13.7 reports the sample distribution of the Iris and CCPP data sets, with respect to their two most relevant respective features—i.e., petal width and length for the Iris data set (fig. 13.7a) and ambient temperature and exhaust vacuum for the CCPP data set (fig. 13.7b).

13.3.1 Classification: the Iris data set

In this case study we exploit PSyKE to extract Prolog rules on a number of classifiers trained on the well-known Iris data set. Notably, the Iris data set contains 150 rows describing as many individuals of the Iris flower. For each exemplary, 4 continuous input features – petal and sepal width and length – are recorded, other than a categorical class label—i.e., which particular sort of Iris plant the exemplary has been classified as. There are three particular sub-sorts of Iris in this data set – namely, Setosa, Virginica, and Versicolor –, and the 150 examples are evenly distributed among them—i.e., there are 50 instances for each class.

[8]`https://archive.ics.uci.edu/ml/datasets/iris` [Last accessed April 17, 2022]

[9]`https://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant` [Last accessed April 17, 2022]

The experimental setting is as follows. First, we train 3 different sorts of classifiers on the Iris data set—namely, a k-nearest-neighbors (kNN), a multi-layer perceptron (MLP), and a decision tree (DT). Then we let PSyKE extract logic rules out of these classifiers using as many extraction procedures. In particular, we rely on REAL, Trepan, and Cart. A portion (50%) of the original data set – namely, the test set – is put aside *before* training to later enable the evaluation of the extracted rule predictive performance.

Accordingly, within the scope of this experiment, we rely on *accuracy* as the preferred metric for both predictive performance and fidelity—where the former measures how good a classifier or the corresponding extracted rules are in classifying Iris instances in absolute terms, while the latter measures the adherence of the extractor output rules w.r.t. the original classifier.

The experiment

Let us assume the Iris data set can be loaded from a CSV file via a script using one of the third-party libraries exploited by PSyKE—i.e., Scikit-Learn or Smile for Python or JVM users, respectively. The Iris data set only contains continuous features. Therefore, Cart is the only algorithm that can be *directly* applied to it, whereas REAL and Trepan can only operate on binary data. Accordingly, PSyKE provides a simple two-step procedure to binarise the data, involving both discretisation and one-hot encoding: a data set can be discretised, one-hot encoded, and split into training and test set via a couple of instructions, providing the percentage of samples to be taken apart from the whole data set to attain the test set. As the next step, we train 4 different classifiers on the training set—namely a kNN, a MLP, and two DT—one with the original, continuous data set, and one with its binarised version[10]. Finally, in the following paragraphs, we show how rules can actually be extracted and what their ultimate shape actually is.

REAL

PSyKE's REAL algorithm can be applied to any BB classifier accepting binary input features—e.g., among the aforementioned models only the DT trained upon the continuous data set is not suitable to apply this extraction technique. The extracted theory is dependent on the training set; different training instances can produce different rules, resulting in slight variations also in the output theory complexity—intended as the number of clauses and terms. An example of REAL applied to a MLP is reported in the following:

```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
```

[10]Code of experiments is available at `https://github.com/psykei/psyke-jvm` and `https://github.com/psykei/psyke-python`

```
    PetalWidth =< 0.78.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalWidth > 0.78, PetalLength in [2.86, 4.91].

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalWidth > 0.78, PetalLength not_in [2.86, 4.91].
```

The theory produces an input space partitioning as reported in fig. 13.8f. It is worthwhile to notice that – especially with more complex data sets – the partitioning could be non-exhaustive—i.e., the logic rules could be unable to classify some samples.

Trepan

PSyKE provides a Trepan algorithm applicable under the same constraint described above for REAL and also its output rules can vary with different training sets. Differently from REAL, Trepan accepts as input 3 optional parameters stating the minimum number of samples to consider for performing further splits (`minExamples`, default: 0), the maximum depth of the produced tree (`maxDepth`, default: 0, i.e. no constraints), and the criterion to adopt for the best split selection (`splitLogic`). At the moment PSyKE can only adopt a default `splitLogic` to assign a binary split to each Trepan node. The method is based on the selection of the most discriminating feature at each split, i.e., the feature having higher probability to produce a tree leaf containing all the samples belonging to a specific class without containing samples of other classes. We plan to implement in the future other methods for *if-then* splits as well as for *M-of-N*splits.

An example of output theory obtained by applying Trepan to a 5-NN is reported in the following:

```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalLength =< 2.28.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalLength > 2.28, PetalWidth not_in [0.78, 1.68].

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor).
```

This theory produces an input space partitioning as reported in fig. 13.8c. In this case, the partitioning is always exhaustive.

Cart

Cart is the third algorithm included in PSyKE to tackle classification tasks. It is directly applicable to DT classifiers and in this case it does not require any extra parameter, since the extraction only relies on the tree structure of the decision tree—that is, all the parameters have to be tuned during the DT creation and training. Otherwise, CART can be applied to other kinds of classifiers, but this implies the creation of an intermediate tree structure and thus to have greater control on the output quality it is advisable to impose a maximum value for the tree depth or the number of leaves.

PSyKE's CART algorithm can operate with both one-hot encoded and continuous input features. In the same way as many other algorithms, CART is able to achieve comparable or even better results when relying on a good discretisation/one-hot encoding technique. In the following an example of theory obtained with CART applied to the DT trained with the continuous Iris data set is reported:

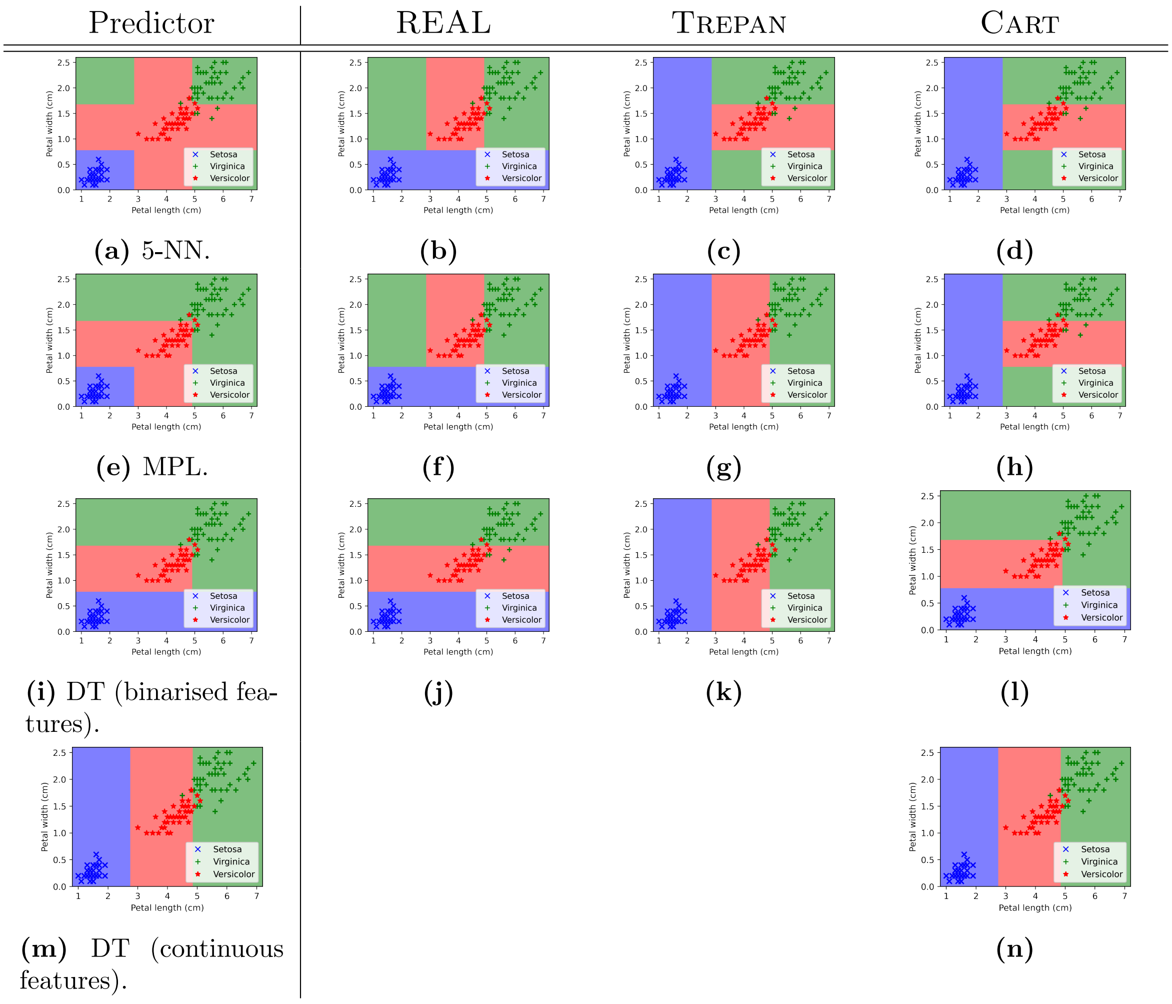
```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalLength =< 2.75.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalLength > 2.75, PetalLength =< 4.85.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalLength > 2.75, PetalLength > 4.85.
```

Figure 13.8n reports the corresponding input space partitioning. The output rules are always exhaustive. The same data set, but previously one-hot encoded, leads to the following theory and to the partitioning reported in fig. 13.8l:

```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalWidth =< 0.78.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalWidth > 0.78, PetalWidth in [0.78, 1.68], PetalLength <= 4.91.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalWidth > 0.78, PetalWidth in [0.78, 1.68], PetalLength > 4.91.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalWidth > 0.78, PetalWidth > 1.68.
```

Results

In the following we report the results of REAL, TREPAN, and CART applied to the Iris data set. All the results are resumed in fig. 13.8 and table 13.2. Column "Predictor" represents the ML step of the process. Accordingly, figs. 13.8a, 13.8e, 13.8i and 13.8m represent the decision boundaries of a 5-NN predictor, a MLP and 2 decision trees, respectively. The first DT is trained with the binarised version of the Iris data set; the other with the original continuous values. Finally, column "Extractor" represents the output of PSyKE. In particular, different extraction procedures – namely, REAL, TREPAN, and CART – are applied to the 4 BB predictors. REAL and TREPAN are not applied to the DT trained with the continuous data set because these extraction procedures cannot handle real-valued input features.



(a) 5-NN. **(b)** **(c)** **(d)**

(e) MPL. **(f)** **(g)** **(h)**

(i) DT (binarised features). **(j)** **(k)** **(l)**

(m) DT (continuous features). **(n)**

Figure 13.8: Comparison between Iris data set input space partitionings performed by the algorithms implemented in PSyKE. Only the two most relevant features are reported—i.e., petal width and length. Missing subplots are due to the impossibility to apply REAL and TREPAN to BB classifiers trained with continuous input features

Table 13.2: Comparison between accuracy and fidelity measurements with different combinations of extraction algorithms and underlying models applied to the Iris dataset

Predictor Type	**Accuracy**	**Extractor Algorithm**	**Fidelity**	**Accuracy**
5-NN	0.94	REAL	0.98	0.95
		Trepan	0.95	0.95
		Cart	0.95	0.95
MLP	0.98	REAL	0.98	0.95
		Trepan	0.98	0.92
		Cart	0.98	0.92
DT (binarised features)	0.98	REAL	0.98	0.97
		Trepan	0.98	0.92
		Cart	1.00	0.98
DT (continuous features)	0.95	Cart	1.00	0.95

It is worth noticing that *(i)* Cart always produces a partitioning equivalent to those of the underlying BB model when the model is a DT, because of its design, and *(ii)* each extractor's output partitioning may be different from other extractors' ones – for instance a procedure may only use the petal width attribute, another procedure may only use the petal length attribute, while other procedures may use both input features, combined in several ways –, but all solutions share a similar predictive performance.

A numerical assessment of the aforementioned predictors and extractors is reported in table 13.2. Values are averaged upon 25 executions, each one with different random train/test splits, but same test set percentage and same parameters for predictors and extractors. The table reports the underlying predictor accuracy as well as the fidelity and accuracy of the extraction procedure. We plan to enhance comparisons between different extractors through fidelity assessments carried out by measuring the decision boundary overlapping regions.

Table 13.2 shows how the Cart extractor always has a fidelity of 1.0 when applied to a DT, since it only inspects the underlying decision tree nodes to build its output rules without any information loss. This implies that PSyKE's Cart extractor is an equivalent (yet explainable) alternative to DT models, as it produces the same output predictions. As for the other extractors, both REAL and Trepan are able to achieve good results in terms of fidelity and accuracy – always above 0.9 in our experiments – in some cases even with a better performance w.r.t. the original model.

13.3.2 Regression: the Combined Cycle Power Plant dataset

In our second case study, PSyKE is exploited to extract Prolog rules out of different BB regressors trained upon the CCPP data set. The data set contains 9568 instances, each one composed of 4 real-valued input attributes – i.e., ambient temperature and pressure, relative humidity and exhaust vacuum – and one real-valued output feature—i.e., the net hourly electrical energy output of the plant.

For this experiment, 3 different regressors are trained on the CCPP data set: a linear regressor (LR), a MLP and a DT. Then, as for the classification case, PSyKE is exploited to extract logic rules out of these BB models by using all the applicable supported procedures—namely, Iter, GridEx and Cart, each one applied to all the aforementioned ML predictors. The data set portion isolated as test set for this experiment is equal to 20%. To assess the predictive performance of BB predictors and extractors as well as the fidelity of extractors w.r.t. the underlying models mean absolute error (MAE) and R^2 score are adopted.

The experiment

The CCPP data set can be loaded from a CSV file. Since it only contains continuous attributes, and since this sort of feature is the only one accepted by PSyKE extractors for regression tasks, no discretisation or binarisation steps are required. However, since the input features have very different ranges of values, a normalisation or standardisation pre-processing is suggested to have more robust BB training phases. For this experiment, a standardisation has been applied to the input features. The operation must be revertible, because extraction techniques applied to BB models trained with standardised (or normalised) data will produce rules containing standardised (or normalised) values. Thus, to obtain human-understandable rules a further inverse transformation is required.

In the following paragraphs PSyKE extractors applicable to BB regressors are described, together with examples of output rules.

Iter

PSyKE's Iter algorithm can be applied to any BB regressor accepting real-valued input features. Iter requires $k+2$ user defined parameters, where k is the number of input dimensions of the data set. The parameters are the hypercube update size for each dimension, the number of starting hypercubes and the similarity threshold used by the extraction procedure to create new cubes instead of expanding the old ones. Also in this case the extracted theory is dependent on the training set, but it also depends on the position, number and dimension of the starting hypercubes. Furthermore, it could be non-exhaustive—especially when the extraction procedure is applied to high-dimensional data sets. An example of input space partitioning produced by ITER applied to a LR is reported in fig. 13.9b.

GridEx

PSyKE's GridEx extractor is applicable under the same constraint described above for ITER. GridEx requires $n+3$ user-defined parameters, where n is the maximum amount of iterations to perform. The other parameters are a similarity threshold, the minimum number of samples to be considered in non-empty hypercubes and the number of slices to perform along each dimension of the hypercubes during each iteration. When the algorithm is applied in adaptive splitting mode, users need to specify the number of partitions to perform on the basis of the importance calculated by GridEx for each input dimension. An example of input space partitioning produced by GridEx applied to a MLP is reported in fig. 13.9g. In this case, the partitioning is always exhaustive. Blank hypercubes correspond to negligible input space regions, so no rules are created to describe these regions.

Cart

CART is the other PSyKE algorithm applicable for regression tasks. All the considerations reported for the classification case in section 13.3.1 hold in this context as well. An example of theory obtained with CART applied to a DT regressor is the following:

```
ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature =< 8.74, Energy is 483.82.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature in [8.74, 11.69], Energy is 476.08.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature in [11.69, 14.45], Energy is 468.94.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature in [14.45, 17.82], Energy is 462.13.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature in [17.82, 22.77], ExhVacuum =< 47.33, Energy is 457.72.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature in [17.82, 22.77], ExhVacuum > 47.33, Energy is 449.16.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature > 22.77, ExhVacuum =< 66.21, Energy is 443.03.

ccpp(Temperature, ExhVacuum, Pressure, Humidity, Energy) :-
     Temperature > 22.77, ExhVacuum > 66.21, Energy is 434.79.
```

Figure 13.9l reports the corresponding input space partitioning. The output rules are always exhaustive.

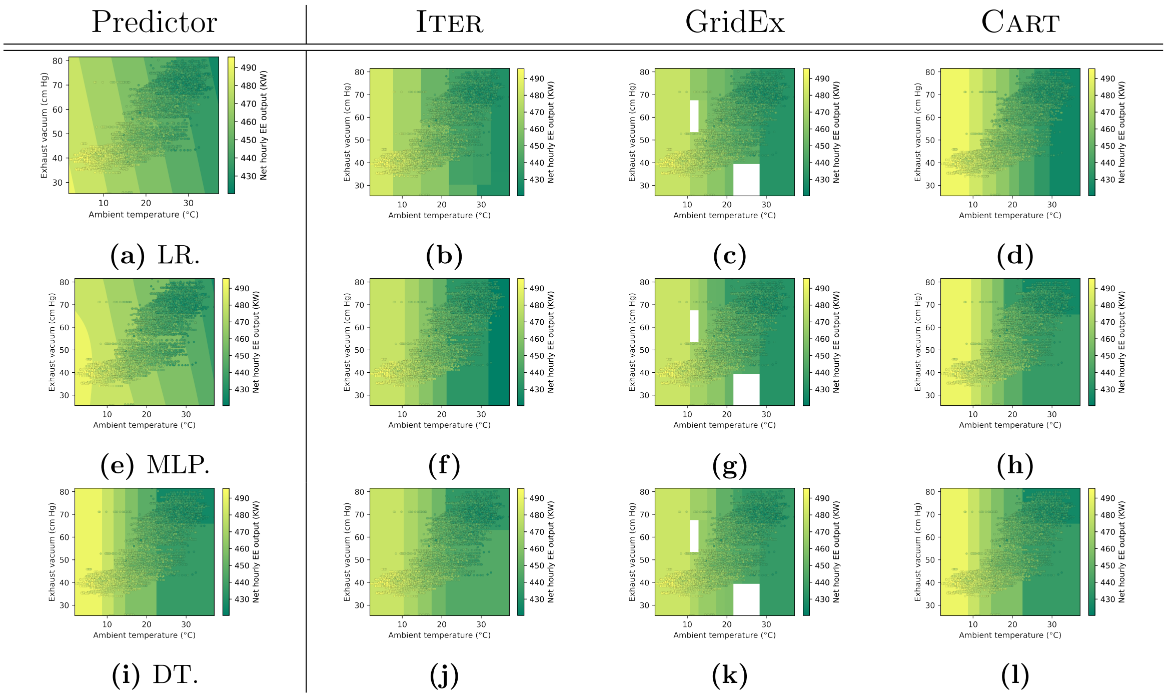


Figure 13.9: Comparison between CCPP data set output predictions provided by the algorithms implemented in PSyKE. Only the two most relevant features are reported—i.e., ambient temperature and exhaust vacuum

Results

The results of ITER, GridEx and CART applied to the CCPP data set are summarised in fig. 13.9 and table 13.3. Column names follow the same logic described for the classification case study in section 13.3.1, so figs. 13.9a, 13.9e and 13.9i represent the decision boundaries of a linear regressor, a MLP and a DT, respectively, while the other figures represent the output of PSyKE extractors. Each one of the extraction procedures suitable for regression tasks is applied to all the aforementioned BB regressors.

Figure 13.9 shows that all the extractors are able to capture the behaviour of the output values w.r.t. the input variables, however one may easily notice that GridEx and CART tend to produce fewer rules than ITER.

The predictive performance of predictors and extractors is assessed in table 13.3. Values are averaged upon 25 executions, each one with different train/test splits, but with the same parameters for both predictors and extractors. All the tested predictors have comparable performance in terms of MAE and R^2 score. Conversely, it is possible to notice that CART and GridEx always appear more reliable than ITER in extracting knowledge out of the underlying predictors.

Table 13.3: Comparison between predictive performance and fidelity measurements – expressed as MAE and R^2 score – with different combinations of extraction algorithms and underlying models applied to the CCPP data set

Predictor			**Extractor**				
Type	**MAE**	**R^2 score**	**Algorithm**	**MAE (data)**	**MAE (predictor)**	**R^2 (data)**	**R^2 (predictor)**
LR	3.62	0.93	Iter	5.47	4.68	0.84	0.88
			GridEx	4.40	2.90	0.89	0.95
			Cart	4.32	2.93	0.90	0.95
MLP	3.73	0.92	Iter	5.03	3.91	0.89	0.92
			GridEx	4.40	2.93	0.89	0.96
			Cart	4.25	2.85	0.90	0.96
DT	3.89	0.91	Iter	4.56	3.06	0.88	0.93
			GridEx	4.04	2.70	0.91	0.95
			Cart	3.89	0.00	0.91	1.00

13.3.3 Discussion

We plan to improve the output theory presentation of PSyKE extractors in several directions. First of all, the semantics of the top-down ordering of Prolog theories could be considered. Rules can be simplified by removing predicates that are always trivially true—i.e., if preceding rules contain dual predicates that are true. For example, second and third rule of the theory reported in section 13.3.1 could be simplified to output a more compact theory:

```prolog
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalWidth =< 0.78.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalLength in [2.86, 4.91].

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica).
```

Both theories are equivalent since the first rule in the original theory implies that in the following rules the `PetalWidth` attribute is always greater than 0.78, whereas the second rule implies that the third rule will be evaluated only if the `PetalLength` attribute has not a value included in the specified interval. This example shows a reduction of 40% of the total number of rule body predicates (from 5 to 3).

Another direction could be to collapse two predicates regarding the same input feature in the same rule body. For example, if a predicate imposes a value greater than a constant for an attribute, and another predicate imposes a value smaller than another constant for the same attribute, the two predicates could be contracted in a unique one by exploiting the semantics of range inclusion. Similarly, if a rule body contains two predicates where the condition represented by one of them is more strict than the one represented by the other, it is possible to remove the less strict predicate. An example can be a rule body with two or more predicates all imposing an attribute greater (smaller) than a specified constant. All the predicates except the one containing the greatest (smallest) constant can be removed without loss of information.

By applying all these rules to the theory regarding the binarised Iris data set presented in section 13.3.1 it is possible to simplify all the rules except the first one and thus to obtain the following theory:

```
iris(SepalLength, SepalWidth, PetalLength, PetalWidth, setosa) :-
    PetalWidth =< 0.78.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, versicolor) :-
    PetalWidth =< 1.68, PetalLength =< 4.91.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalWidth =< 1.68, PetalLength > 4.91.

iris(SepalLength, SepalWidth, PetalLength, PetalWidth, virginica) :-
    PetalWidth > 1.68.
```

This theory can be further simplified by applying the first strategy described in this section, i.e., by substituting the third and fourth rules with a unique rule having as body `true`. The resulting theory will have only 3 rules with 4 predicates in their bodies, whereas the starting one has 4 rules and 9 predicates.

13.4 Recap and Research Perspectives

In this chapter we present the design of PSyKE, a new general-purpose platform supporting symbolic knowledge extraction from opaque ML predictors. PSyKE offers many comparable and interchangeable extraction procedures providing as output first-order logic clauses. It can be exploited in the majority of supervised learning tasks—i.e., classification and regression tasks.

In the future we plan to enrich PSyKE with other state-of-the-art extraction algorithms, comparison metrics between the implemented procedures, and other utilities—i.e., discretisation strategies. We also plan to explore other formalisms to present output rules – e.g. the ProbLog syntax to introduce the concept of probabilistic rule –, as well as other representations and extraction procedures which are better suited to manage data sets involving a wide number of features.

From a research perspective, we aim at further investigating the effectiveness of PSyKE in running EU projects, like StairwAI[11] and EXPECTATION [CCN+21].

StairwAI is an H2020 project aimed at providing a service layer for the AI-on-demand platform,[12] with the purpose of aiding both individual and companies to *(i)* find the most adequate AI asset for their needs – requiring mapping of use cases to proper AI assets –, and *(ii)* experimenting selected AI assets on custom data and on specific problems, using the platform itself—thus requiring tools for predicting the hardware resources needed for running the corresponding software.

[11] `https://cordis.europa.eu/project/id/101017142` [Last accessed April 17, 2022]

[12] `https://cordis.europa.eu/project/id/825619` [Last accessed April 17, 2022]

Expectation is a CHIST-ERA IV project[13] aimed at exploring the provisioning of *personalised* explanations for ML techniques by combining SKE and multi-agent-based negotiation and argumentation. There, symbolic knowledge is expected to act as the *lingua franca* among many heterogeneous ML-based predictors – possibly trained on different data sets, via different algorithms, at different locations –, hosted by as many software agents. Personalisation and predictive accuracy are therefore attained by combining the symbolic knowledge extracted by several agents. The combination takes advantage of negotiation and argumentation techniques, possibly involving the users themselves.

Both projects massively rely on sub-symbolic AI, and in both cases the need of making sub-symbolic knowledge explainable is prominent. PSyKE could then be applied to extract logic rules and reveal information about the path that leads to a certain prediction—in the explanation perspective. Since PSyKE currently works as a distiller of knowledge, further investigation will be devoted to the explanation of a single outcome (prediction of a model). Moreover, it could be interesting to compare results with those obtained by directly learning a symbolic model.

[13] `https://www.chistera.eu/projects/expectation` [Last accessed April 17, 2022]

Chapter 14

Enriching the Ecosystem with Probabilistic Logic Programming

This chapter contains contributions from the following works of ours: [DCC21]

This chapter contains contributions from the following Master's thesis: [Del21], which we supervised

Artificial Intelligence (AI) is progressively conquering the software industry to become one of the most pivotal fields, with a fast-paced evolution of challenges and requirements that existing technologies often fail to match. Accordingly, the increasing demand for transparent and pervasive intelligence is opening new horizons for logic programming (LP) and symbolic AI approaches [CCMO21a, COS21, OFM$^+$21]. However, logic-based approaches alone are often not suitable to be integrated with present-day planning and learning workflows, which natively deal with uncertainty and probabilistic decision-making [CCDO20, CCO20].

Probabilistic logic programming (PLP) [NS92, Rig18] is a research field that investigates the combination of LP with the probability theory (cf. section 4.2.1). State-of-the-art PLP solutions [DRKT07, Rig07] have reached a considerable level of maturity and theoretical reach. Not only has *exact* probabilistic resolution been reified into actual programming languages, but also approximate resolution, and learning of probabilities from data. However, existing technologies currently rely upon monolithic runtimes, often targeting single platforms or having inconvenient constraints and dependencies [KDDR$^+$11, NZRS12]—limiting their interoperability and portability with mainstream programming platforms. This follows a general tendency of logic-based technologies, which are often constructed as technological silos – being so optimised for performance and correctness while being poorly interoperable among each other – targetting the LP community alone.

To overcome such tendency towards the creation of isolated monoliths, the notion of *logic ecosystem* [CCO21a] has recently been proposed. There, the authors argue that LP facilities – e.g. knowledge representation, unification, clauses indexing, resolution, etc. – should be made independently available to the widest possible audience—there including mainstream developers and logic programmers, and all major programming platforms. Most notably, LP facilities should not only be exploitable as stand-alone applications (e.g. Prolog interpreters) but also (and foremost) as libraries—thus enabling re-use at the mechanism level. In this perspective, logic ecosystems consist of *extensible* technological *frameworks* where single LP facilities can be incrementally constructed on top of the previous ones, other than used—either individually or composedly. Notably, the authors in [CCO21a] prose 2P-Kt as the technological reification of a logic ecosystem. Unfortunately, however, PLP is not among the LP facilities currently supported by 2P-Kt.

Accordingly, in this work we propose an extension of the 2P-Kt ecosystem aimed at supporting PLP via an ad-hoc implementation of the ProbLog language. The proposed implementation aims at overcoming the interoperability and portability issues of state-of-the-art PLP solutions. In fact, as part of the 2P-Kt ecosystem, our ProbLog implementation can be compiled/run on several strategic platforms, other than used as a library in multiple programming languages. Our solution provides PLP support on top of standard Prolog solvers. Hence, as a side contribution, we provide insights about how a ProbLog solver can be realised on top of Prolog's SLD(+NF) resolution principle.

It is worth highlighting how our current goal is to provide a usable and functioning PLP code base, initially supporting only the fundamental features, and aiming to be flexible for future growth. Outperforming existing solutions is not amongst our primary concerns. Conversely, we aim to open the horizons for wider adoption of LP and PLP, by favouring portability and by making it easier to exploit from outside the LP realm. In this regard, we describe a number of examples aimed at demonstrating the usability and portability of our PLP solution on multiple runtimes and programming platforms.

14.1 State-of-the-art technologies for PLP

As discussed in section 4.2.1, PLP commonly represents probabilistic theories as LPAD (Logic Programs with Annotated Disjunctions) [VVB04], where clauses admit disjunctions of atoms in their heads, and each atom is labelled with a probability value. Meaning is provided via Sato's distribution semantics [Sat95, SK97]. Implementations may rely upon binary decision diagrams (BDD) [Ake78, LMS14] (or their variants/extensions) to make probabilistic inference more efficient, following the knowledge compilation approach [BR13, VRVdBDR14].

A number of programming languages follow the LPAD approach over the distribution semantic, there including ProbLog and `cplint`. They both rely on (some variant of) BDD to support probabilistic reasoning. Within the scope of this chapter, we consider them as interesting solutions for PLP as they come with some actually usable technology. In the reminder of this section, we briefly analyse ProbLog and `cplint` from a technological perspective.

ProbLog. ProbLog [DRKT07] is a probabilistic programming language providing PLP support on top of Prolog. We appreciate the simplicity of the language and the high compatibility with traditional Prolog—hence why we target a ProbLog extension for 2P-Kt. ProbLog, in particular, leverages upon a number of aspects of Prolog's operation to attain PLP support. First, it relies on knowledge compilation of annotated facts into ordinary Prolog clauses. Then, it exploits Prolog's backtracking mechanism to enumerate the possible worlds in which a query is true. These are called 'explanations' in PLP's nomenclature, while they are ordinary solutions in the eyes of a Prolog solver. Finally, ProbLog attempts to iteratively build a BDD as part of the resolution process, in order to keep the problem of computing the probability of a query tractable. The Prolog solvers' dynamic KB are used as ancillary data stores in the meanwhile. Once all the possible worlds have been enumerated, the resulting BDD is fully navigated to efficiently compute the probability of the query.

Currently, the ProbLog project consists of a Python codebase, depending on a number of native libraries and tools—such as the YAP Prolog technology [CRD12]. Such technological choices limit the portability of ProbLog outside the scope of the major desktop operative systems (e.g. Windows, Linux, or Mac OS). Notably, this issue is mitigated by the existence of a publicly-available Web application letting users experiment ProbLog from their browsers. In any case, to the best of our understanding of the ProbLog's documentation and source code, ProbLog is mainly intended as a stand-alone command-line application and interpreter, and its usage as a library is not explicitly supported.

`cplint`. The `cplint` system (CPLogic INTerpreter) [Rig07] applies knowledge compilation to logic programs annotated à la CP-Logic [VDB09]. Notably, it compiles probabilistic clauses into Multivalued Decision Diagrams (MDDs) [TDD78], an extension of BDDs. Thus, differently from ProbLog, the random variables corresponding to logic clauses can be multi-valued. Furthermore, `cplint`'s probabilistic programs support negated atoms.

`cplint` leverages upon a Prolog meta-interpreter to solve probabilistic queries. Similarly to ProbLog, it keeps track of the solutions encountered during resolution, while simultaneously building a MDD aimed at leter being able to draw probabilities.

Currently, the `cplint` project consists of a Prolog codebase targetting the SWI-Prolog [WSTL12] platform. Such technological choices limit the portability of `cplint` on platforms for which SWI-Prolog is not available, or platforms that are poorly interoperable with (SWI-)Prolog—e.g. Android, the JVM or iOS. Notably, this issue is mitigated by the existence of a publicly-available Web application letting users experiment `cplint` from their browsers. In any case, to the best of our understanding of its documentation and source code, `cplint` is mainly intended as a stand-alone command-line application and interpreter, or as a Prolog library.

14.1.1 Logic Ecosystems and 2P-Kt

The current practice of logic-based technologies (LBT) follows a tendency where software contributions are constructed as extensions or on top of the Prolog language, often on native (i.e. based on C or C++) technologies. Such a tendency has pushed the LP community towards a situation where tools consist of poorly interoperable technological silos, where: *(i)* logic facilities (e.g. unification; clauses storage, indexing, or retrieval; resolution, etc.) are not adequately separated, and can only be exploited by means of Prolog, *(ii)* usage of logic facilities must step through a stand-alone application (commonly, either graphical or command-line), as they are not available "as a library" to other programming platforms *(iii)* the portability of LBT technologies is constrained on the platforms the underlying Prolog system supports.

To overcome such issues the 2P-Kt technology has been recently proposed in [CCO21a], along with the notion of logic *ecosystem*. There 2P-Kt is considered as an ecosystem of loosely coupled *modules*, each one dedicated to a single logic facility. Hence, overall, it consists of a collection of logic facilities, exposed to the developers as *multi-platform* libraries—and, possibly, as stand-alone applications as well. There, multi-platform support aims at letting mainstream programming platforms benefit from the sole logic facilities they need, natively—and without having to interact with a full fledged Prolog system.

Arguably, multi-platform support is fundamental to let researchers and practitioners from the many branches of computer science and artificial intelligence benefit from LBT. Along this line, we believe logic facilities – such as probabilistic resolution – should be exploitable on mainstream programming platforms and languages – e.g. JVM, Python, JavaScript, etc. – to ease the exploitation of LP for the niches by which those platforms and languages are used the most. On the long run, for instance, we hope that bringing LP on Python will ease its hybridisation with data science, while bringing it on JavaScript will ease its hybridisation with the Web, and so on.

Accordingly, 2P-Kt currently explicitly targets the Kotlin, Android, JVM, and JavaScript platforms, while other platforms – such as iOS and Python – are going to be supported soon, thanks to the multi-platform programming facilities offered

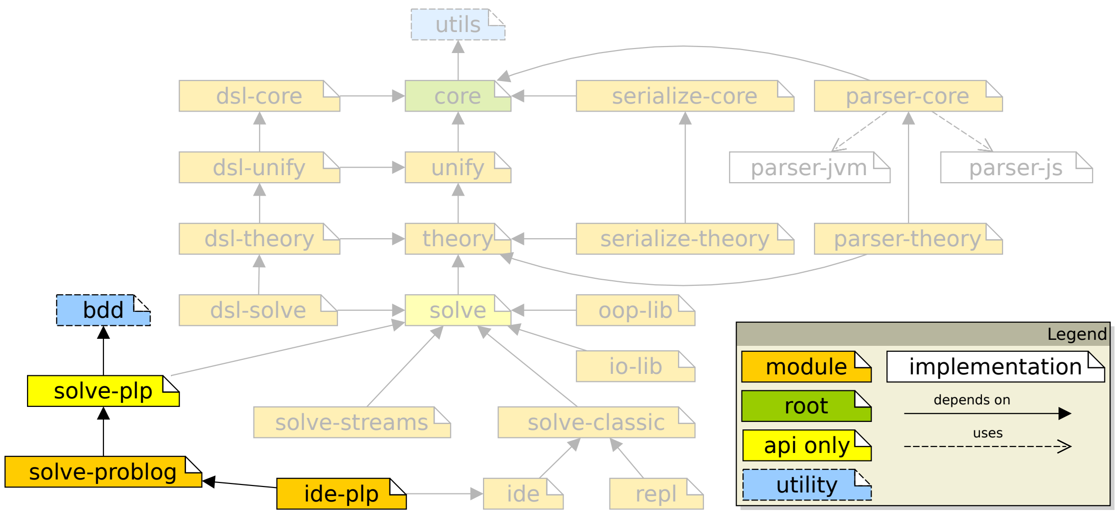


Figure 14.1: Architectural overview of our PLP and ProbLog modules, and their role within the 2P-Kt ecosystem

by Kotlin[1]. Of course, we acknowledge that different languages and platforms may follow different conventions and paradigms. Hence, multi-platform must not be realised via mere cross-compilation on several platforms, but rather ad-hoc software layers should be provided to harmonise LP to the target platforms, at the paradigm level (cf. [CCS+20]).

2P-Kt currently focuses on supporting knowledge representation and automatic reasoning via logic programming. The modular, unopinionated architecture of 2P-Kt is deliberately aimed at supporting and encouraging extensions towards other sorts of symbolic AI systems than Prolog—including PLP, which is currently missing. Accordingly in the following, we discuss how a module for ProbLog can actually be designed and realised to enrich the 2P-Kt ecosystem.

14.2 Design of Probabilistic Solver Module

Here we discuss how the 2P-Kt ecosystem can be enriched to support PLP. In particular, our goal is to add two major facilities to the ecosystem, namely: *(i)* a general-purpose API for probabilistic resolution, and *(ii)* a purpose-specific API for ProbLog-like resolution. Of course, while pursuing this purpose, the underlying technical requirement is to re-use the pre-existing facilities offered by 2P-Kt as much as possible. This includes terms, clauses, and theories representation, as well as Prolog's SLDNF resolution.

[1] `https://kotlinlang.org/docs/mpp-supported-platforms.html`

Accordingly, as depicted in fig. 14.1, PLP support is injected into the ecosystem via multiple self-contained and inter-dependent modules, each one representing a contribution of our proposal. Arrows indicate direct dependencies from one module to another. Of course, dependencies are *transitive*, meaning that each module inherits (and can therefore exploit) all the facilities carried by the other modules it depends upon, either directly or indirectly. Notably, PLP related modules are: `:bdd`, `:solve-plp`, `:solve-problog`, and `:ide-plp`.

The `:bdd` module represents our proposal for the binary decision diagram manipulation library. This module is purely self-contained, in the sense that it does not rely upon any external facility to support BDD. Rather, it consists of a pure Kotlin solution, which therefore puts no additional constraint on the platforms targetted by 2P-Kt. It is worth noting that, with such a choice, we intend to promote the usage of the library as a lean external dependency on other projects as well.

The `:solve-plp` module is meant to bundle all the entities and traits that are common to any potential implementation of solvers for the PLP paradigm. In other words, it is where our goal *(i)* is realised. This is a purely abstract module, that only provides API, interfaces and classes on which multiple PLP solver implementations can rely upon. Notable, this module depends on 2P-Kt's `:solve` module, which provides common abstractions for logic solvers and fixes their API, in order to keep them interoperable. In other words, we model *probabilistic* logic solvers as a direct subset of logic solvers.

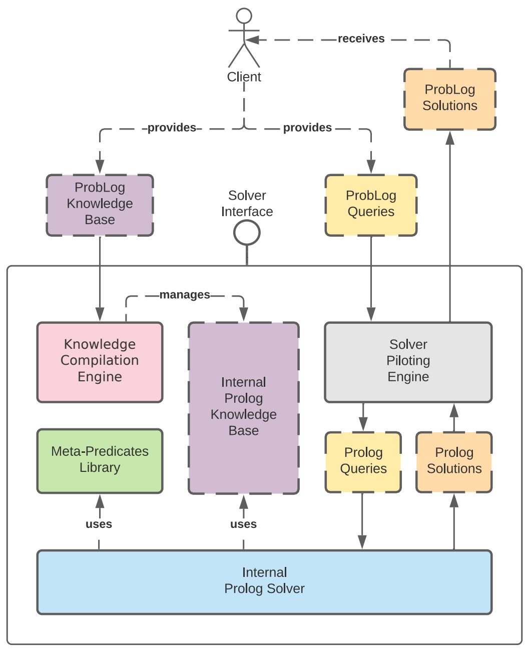
The `:solve-problog` module contains the actual implementation of the PLP solver supporting the ProbLog language. In other words, this is where our goal *(ii)* is realised. As ProbLog solvers will be particular cases of probabilistic solvers, the `:solve-problog` module depends on the abstractions of `:solve-plp` and it is compliant to them. The other fundamental (and indirect) dependency is the `:bdd` module, which is used for manipulating binary decision diagrams during probabilistic logic goal resolution. Additionally, it also depends on `:solve-classic`—as ProbLog solvers will exploit ordinary Prolog resolution behind the scenes. Further details about the inner design and functioning of this module are discussed in the remainder of this section, and represent the main contribution of this chapter.

Finally, the `:ide-plp` module implements a stand-alone graphical application based on JavaFX, aimed at letting 2P-Kt users practice with ProbLog via an integrated environment.

14.2.1 Design Rationale

Figure 14.2a provides an overview of the overall design of our `:solve-problog` module. Overall, the module aims at providing a notion of ProbLog solver as a particular case of logic solvers. As any other sort of solver in 2P-Kt, ProbLog

Figure 14.2: Architecture of our ProbLog solver (left), with a focus on the KB recompilation step (right)



(a) Architecture and information flow of our ProbLog solver

```
male(john).
0.80::male(mike).
0.65::female(anna).
0.60::parent(mike, john).
0.95::father(X, Y) :- male(X), parent(X, Y).

                ||
           automatic
         recompilation
                ⇓

prob(E, male(john)) :- expl_build(E, 1.0).
prob(E, male(mike)) :- expl_build(E, 0.8).
prob(E, female(anna)) :- expl_build(E, 0.65).
prob(E, parent(mike, john)) :- expl_build(E, 0.6)
prob(E, father(X, Y)) :-
   expl_build(E0, 0.95),
   prob(E1, male(X)), prob(E2, parent(X,Y)),
   expl_and(E, [E0, E1, E2]).
```

(b) Example of KB recompilation

solvers accept users' queries as inputs – consisting of (possibly partially instantiated) logic atoms – and produce a multitude of solutions as outputs—consisting of variable assignments and probabilities. Notably, solutions are computed against a ProbLog knowledge base, which, in practice, consists of a Prolog theory with annotated clauses.

To perform *probabilistic* resolution, each ProbLog solver relies on a Prolog solver behind the scenes. The Prolog solver expects the probabilistic theory to be compiled into an ordinary Prolog theory aimed at constructing a BDD as the resolution process proceeds. In this phase, each probabilistic clause of the form *p*`::`***`Head`*** `:-` ***`Body`*** is transformed into and ordinary Prolog clause of the form `prob(`***`Explanation`***`,` ***`Head`***`) :-` ***`Body2`***, where ***`Explanation`*** represents the BDD to be constructed out of the probability *p* and ***`Body`***, whenever the probability of some sub-goal ***`Head`*** must be computed. A number of *ad-hoc* meta-predicates can be exploited in the clauses' bodies to serve the purpose of incrementally building a BBD. Under such assumption, the underlying Prolog solver may answer to probabilistic queries of the form `prolog_query(-`***`Probability`***`, +`***`Goal`***`)`. More precisely, the `prolog_query/2` predicate is in charge of *(i)* computing all possible

Prolog solutions for ***Goal*** and *(ii)* constructing their specific BDD, then *(iii)* merging them into a unique BDD aimed at computing the overall ***Probability*** of ***Goal***.

To sum up, a ProbLog solver is a bi-directional façade among the user and the underlying Prolog solver. It takes care of translating probabilistic theories and queries in Prolog form, and Prolog solutions back into probabilistic form. Given this overview, the design of our PLP solver is built on top of three interconnected components: *(i)* a knowledge compilation engine, *(ii)* a library of meta-predicates, and *(iii)* a solver piloting engine. In the remainder of this section, we delve into the details of these components.

Knowledge Compilation Engine

Each ProbLog solver of ours is backed by a Prolog solver aimed at computing an *explanation* (i.e. a BDD) for each possible probabilistic query. However, the Prolog solver can only deal with ordinary logic theories consisting of unannotated Horn clauses. Accordingly, *knowledge compilation engine* is the architectural component in charge of converting annotated probabilistic theories provided by the ProbLog users into ordinary Prolog users. It does so by applying a number of rewriting rules to the probabilistic theory:

$$
\begin{array}{rcl}
[\![f(\bar{X})\texttt{.}]\!] & \longrightarrow & \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl\_build(}E\texttt{, 1.0).}\text{'} \\
[\![p\texttt{::}f(\bar{X})\texttt{.}]\!] & \longrightarrow & \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl\_build(}E\texttt{, } p\texttt{).}\text{'} \\
[\![p\texttt{::}f(\bar{X}) \texttt{ :- } b_1(\bar{X}_1)\texttt{, ..., } b_n(\bar{X}_n)\texttt{.}]\!] & \longrightarrow & \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl\_build(}E_0\texttt{, } p\texttt{),} \\
 & & \quad \texttt{prob(}E_1\texttt{, } b_1(\bar{X}_1)\texttt{), ..., prob(}E_n\texttt{, } b_n(\bar{X}_n)\texttt{),} \\
 & & \quad \texttt{expl\_and(}E\texttt{, [}E_0\texttt{, }E_1\texttt{, ..., }E_n\texttt{]).}\text{'} \\
[\![p_1\texttt{::}f_1(\bar{X}_1)\texttt{, ..., } p_m\texttt{::}f_m(\bar{X}_m) \texttt{ :- } \bar{b}\texttt{.}]\!] & \longrightarrow & \text{`}[\![p_1\texttt{::}f_1(\bar{X}_1) \texttt{ :- } \bar{b}\texttt{.}]\!]\texttt{. } \ldots [\![p_m\texttt{::}f_m(\bar{X}_m) \texttt{ :- } \bar{b}\texttt{.}]\!]\texttt{.}\text{'}
\end{array}
$$

There, the first rule handles the case of unannotated facts (a.k.a. evidence). They are considered as certain facts—i.e. facts having 1.0 as probability. The second rule handles the case of annotated facts having a probability $p \in [0, 1] \subset \mathbb{R}$. Finally, the third rule handles the case of annotated rules, whereas the last rule handles the case of probabilistic clauses having annotated disjunctions in their heads. Because of space limitations, we here omit other rules aimed at handling conjunction, negation, or implication in clauses' bodies. In all such cases, $f, f_1, \ldots, f_m, b_1, \ldots, b_n$ denote logic predicates' symbols of arbitrary arity, $p, p_1, \ldots, p_m$ are real numbers in the $[0, 1]$ range denoting probability values, $\bar{X}, \bar{X}_1, \ldots, \bar{X}_n, \bar{X}_m$ denote tuples of logic terms of arbitrary length, while $\bar{b}$ denote a conjunction of logic atoms involving zero, one, or more atoms.

Figure 14.2b exemplifies the knowledge compilation engine in action on a simple probabilistic theory. As the reader may notice, the resulting Prolog theory consists of a number of rules of the form `prob(`***-Explanation, +Goal***`)`, aimed at computing the an ***Explanation*** for a particular ***Goal***. The bodies of such rules may exploit a number of built-in meta-predicates aimed at iteratively constructing an explanation out of simpler explanations.

Library of Meta-Predicates

A fundamental prerequisite for the knowledge compilation engine to work is that BDD can be suitably represented in logic, as explanations, at the end of the day, consist of BDD instances.

To address such a need, in our ***:solve-plp*** module, we define a whole new class of logic constants aimed at *referencing* particular instances of BDD. So BDD instances – which are in-memory data structures in Kotlin's object-oriented world – are treated as constants in the logic realm. In this way, BDD instances can be carried around, constructed, or composed as part of resolution, and possibly bound to variables such as ***Explanation*** or E, E_1, ..., E_n mentioned above.

To make it possible to create and compose BDD from a logic program, we introduced a library of Prolog-compliant meta-predicates, each one implementing a specific task supporting ProbLog-like inference. Of course, as such meta-predicates operate on data structures that lay outside the logic realm, they cannot be defined in Prolog. Accordingly, through the *generator* mechanism of 2P-Kt (cf. [CCO21b]), we are able to implement the behaviour of these meta-predicates with object-oriented Kotlin code. At the functional level, however, the behaviour of most relevant meta-predicates can be described as follows:

`expl_and(-E, [+E₁, ..., +Eₙ])` — provided that variables E_1, ..., E_n are bound to as many constants representing $n \geq 2$ BDD, this meta-predicate merges them all into a new BDD representing their conjunction, and binds a constant to E referencing that BDD

`expl_or(-E, [+E₁, ..., +Eₙ])` — like the above, but for disjunction

`expl_not(-E, +E')` — like the above, but for negation

`expl_build(-E, +P)` — provided that variable E is bound to a number representing a valid probability value, this meta-predicate creates a bare new, minimal BDD out of that probability value, and binds a constant to E referencing that BDD

The `prolog_query(-Probability, +Goal)` meta-predicate then closes the loop, acting as the main entry point for probabilistic resolution in Prolog. The first argument represents the numeric probability of the goal being queries, and the second argument is the goal itself. The probability argument can either be an input number or an output variable. If the goal argument is a non-ground term, its variables are substituted for each solution found by the solver. Of course, despite

```
0.6::edge(1,2).
0.1::edge(1,3).
0.4::edge(2,5).
0.3::edge(2,6).
0.3::edge(3,4).
0.8::edge(4,5).
0.2::edge(5,6).

path(X,Y) :- edge(X, Y).
path(X,Y) :- edge(X, Z),Y \== Z,path(Z, Y).
```

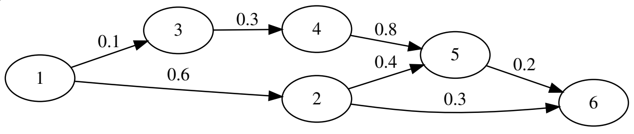


Figure 14.3: Example of Probabilistic Graph Modeling: ProbLog syntax (left) and corresponding graph (right)

the `prolog_query/2` meta-predicate simulates a lazy enumeration of solutions via backtracking, the whole set of solutions must be eagerly computed behind the scenes, in order to compute probabilities. Hence, queries having an infinite proof tree may lead to a situation where the solver gets stuck or saturates the available memory even before the first solution is presented to the user.

Solver Piloting Engine

The last piece needed by our system to fully implement a PLP inference solver is a component aimed at hiding the presence of an underlying Prolog solver. We call this component the *solver piloting engine*.

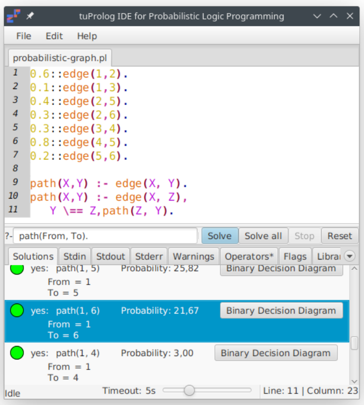
This component is responsible for accepting LP and PLP queries from clients, properly configuring the underlying LP solver, piloting it to infer the solutions, extracting the probability values and presenting the results. As a matter of fact, it represents the presentation layer of our system. Notably, solver configurations are handled at this level, and the component is capable of passing both LP and PLP queries to the inner solver on demand.

Also, the solver piloting engine recompiles queries and goals bidirectionally to be compliant with the meta-predicates semantics of our solution. For instance, our solution assumes that each query is represented via the `prob_query/2` predicate. Considering the example in fig. 14.3, a query such as `path(`*`From`*`, `*`To`*`)` would be transformed in `prob_query(`*`P`*`, path(`*`From`*`, `*`To`*`))`. Once solutions are found, the *solver piloting engine* extracts the two terms *`P`* and `path(`*`From`*`, `*`To`*`)`, and presents their values to the clients in the correct format.

14.3 Multi-platform Support Demonstration

Here we provide a demonstration of our ProbLog module for 2P-Kt. More precisely, we show how our solution supports: *(i)* a wide gamma of usage modalities – ranging from issuing ProbLog queries via a GUI to usage "as a library" –, and

(a)



```
0.6::edge(1,2).
0.1::edge(1,3).
0.4::edge(2,5).
0.3::edge(2,6).
0.3::edge(3,4).
0.8::edge(4,5).
0.2::edge(5,6).

path(X,Y) :- edge(X, Y).
path(X,Y) :- edge(X, Z),
   Y \== Z,path(Z, Y).
```

```
?- path(From, To).
```

```
yes:  path(1, 5)     Probability: 25,82
        From = 1
        To = 5
yes:  path(1, 6)     Probability: 21,67
        From = 1
        To = 6
yes:  path(1, 4)     Probability: 3,00
        From = 1
        To = 4
```

(b)

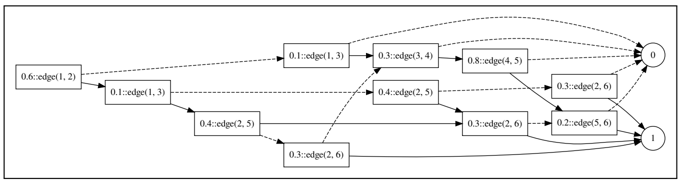


Figure 14.4: 2P-Kt PLP IDE (14.4a) and corresponding BDD built by the solver (14.4b)

(ii) a number of mainstream programming platforms and languages.

In particular, our demonstration works by solving a probabilistic query against the trivial probabilistic logic program from fig. 14.3 – where a probabilistic graph is modelled in ProbLog –, enumerating all possible solutions and interpreting them as possible paths and their probabilities. We perform this action multiple times, and in several ways, each time showing a different usage modality. Notably, we exemplify the usage ProbLog as a JavaFX-based graphical application, other than as a Kotlin, Java, Android, Python, and JavaScript library.

Figure 14.4 shows 2P-Kt's IDE, tailored on our ProbLog module. The whole demonstration can be reproduced by downloading the PLP IDE executable (`2p-ide-plp-X.Y.Z-redist.jar`) from `https://github.com/tuProlog/2p-kt/releases/latest`. The IDE accepts ProbLog theories as input, either from a file or as bare textual input, and it is designed to resemble a simple text editor. One can issue a query and submit it to the underlying ProbLog solver. Once computed, solutions to that query are shown in a list view. Also, a tab view enables the inspection of the internal state of the solver. Figure 14.4b depicts the BDD used by our ProbLog solver behind the scenes while computing the probability of solution `path(1, 6)`. Notably, BDD representation is yet another function of our IDE, attained via an automatically-generated DOT [GN00] specification.

Figure 14.5 shows how our ProbLog module can be used "as a library" on multiple programming platforms and languages, namely Kotlin (for both the JVM and Android platforms), Python, and JavaScript. The Java language is supported

```kotlin
// Kotlin
val clausesParser = ClausesParser.withOperators(PROBLOG_OPERATORS)
val probabilisticTheory = clausesParser.parseTheory("⟨theory from fig. 14.3⟩")
val problogSolver = Solver.problog.solverWithDefaultBuiltins(staticKb = probabilisticTheory)
val goal = Struct.of("path", Var.of("From"), Var.of("To"))
for (solution in problogSolver.solve(goal, SolveOptions.allLazily().probabilistic()))
    if (solution.isYes)
        println("yes: ${solution.solvedQuery} with probability ${solution.probability}")
```

```python
# Python
probabilisticTheory = parse_theory("⟨theory from fig. 14.3⟩", PROBLOG_OPERATORS)
problogSolver = problog_solver(static_kb=probabilisticTheory)
query = struct('path', var('From'), var('To'))
for solution in problogSolver.solve(query, solve_options(lazy=True, probabilistic=True)):
    if solution.is_yes:
        print(f"yes: {solution.solved_query} with probability {probability(solution)}")
```

```javascript
// JavaScript
let clausesParser = ClausesParser.Companion.withOperatorSet(PROBLOG_OPERATORS)
let scope = Scope.Companion.empty()
let probabilisticTheory = clausesParser.parseTheory("⟨theory from fig. 14.3⟩")
let problogSolver = Solver.Companion.problog.solverWithDefaultBuiltinsAndStaticKB(probabilisticTheory)
let query = scope.structOf("path", [scope.varOf("From"), scope.varOf("To")])
let options = probabilistic(SolveOptions.Companion.allLazily())
let si = problogSolver.solveWithOptions(query, options).iterator()
while (si.hasNext()) {
    let solution = si.next();
    if (solution.isYes)
        console.log('yes: ${solution.solvedQuery} with probability ${probability(solution)}')
}
```

Figure 14.5: Usage of 2P-Kt's ProbLog module "as a library" on multiple programming languages

as well, despite not being depicted in the figure. The similarity among the code snippets is deliberate and aimed at stressing how the many 2P-Kt ports share a common design and API, despite the slight syntactical differences characterising the target language. The conceptual flow is analogous: *(i)* a `ClausesParser` is instantiated out of the set of ProbLog predicates (i.e. Prolog's standard predicates, plus `::/2`), *(ii)* it is then used to parse the ProbLog program from fig. 14.3, *(iii)* the resulting `Theory` is used as *static* KB of a newly instantiated ProbLog `Solver`, *(iv)* the query `path(`*`From,To`*`)` is programmatically constructed, and *(v)* issued to the `Solver`, as a *probabilistic* query. Solutions are then *(vi)* enumerated, and, finally, *(vii)* positive solutions are printed, along with their *probabilities*. For the sake of reproducibility, the provided snippets can be executed on all the supported platforms by cloning the Git repository `https://github.com/tuProlog/2pkt-problog-compatibility-demo`, and by following the contained instruction. As the reader may easily observe, the resulting solutions and probabilities are the same depicted in fig. 14.4a.

14.4 Recap and Research Perspectives

This chapter describes the design and implementation of a ProbLog solver as a module of a logic ecosystem. The extension pursues the twofold goal of *(i)* enriching the 2P-Kt logic ecosystem and technology towards PLP and, in particular, ProbLog, and *(ii)* bridging PLP and main-stream programming platforms and languages by letting developers benefit from a *library* providing probabilistic reasoning capabilities to their projects.

The proposed solution is still in its infancy, and it is still not suitable to be compared with other proposals in the field—at least for what concerns performance or feature richness. However, by working on top of the 2P-Kt ecosystem, our solution inherits large platform support – as demonstrated in this chapter –, thus overcoming the usability and portability constraints that affect other solutions in this field. In fact, our technology can be deployed on all the platforms supported by 2P-Kt—which currently include, but are not limited to, the JVM, Android, Python, and JavaScript. In the long term, we believe such technological openness will play a fundamental role in bringing the benefits of (P)LP to the general public and letting AI practitioners exploit (P)LP with minimal effort. In this perspective, our proposal represents a first step in this direction.

Ultimately, one of the top priorities of this research effort is to leave the door open to future developments. Our design is purposely abstract, and we endorse the future exploration of alternative implementation ideas. Among the others, we envision future directions involving: approximate inference support, more efficient knowledge compilation data structures, or the exploitation alternative resolution strategies such as the tabled or concurrent ones—other than, of course, comparative benchmarks aimed at assessing our solutions w.r.t. the state of the art.

Chapter 15

Enriching the Ecosystem: the Future

In this chapter we discuss a number of research directions stemming from, building upon, or overlapping the contribution of this thesis. Some initial steps have been already performed along all such directions, by either us or the students we have (co-)supervised. Hence, for each research direction, we briefly provide *(i)* some background or a brief overview of the literature of the field, *(ii)* a discussion about why the field is interesting w.r.t. the scope and the goals of this thesis, *(iii)* some insight about how that research direction fits the 2P-KT ecosystem (or vice versa), and *(iv)* a few reference about what we or our students have already done along that direction.

Accordingly, in the reminder of this chapter we discuss the following research lines, in no particular order: concurrent logic programming and its role in speeding up logic or hybrid computations (section 15.1); graph neural networks and their role in the sub-symbolic processing of symbolic knowledge (section 15.2); symbolic knowledge *injection*, its role within XAI, and its duality w.r.t. symbolic knowledge extraction (section 15.3); tuple-based coordination and its role in the interaction of intelligent agents among the Internet (section 15.4); and finally inductive logic programming and its role into the 2P-KT ecosystem (section 15.5).

15.1 Concurrent Logic Programming

This section contains contributions from the following Master's thesis: [Gio21], which we supervised

Here we present the fundamental notions of *concurrent* logic programming, and their potential role in the speed up of logic resolution. This is a very relevant topic within the LP community itself, and, as we further discuss in this section, it has the potential to impact hybrid systems as well—i.e. systems combining or integrating LP and sub-symbolic AI.

15.1.1 Brief overview of the field

One of the most attractive features of LP is the clean separation of logic and control [Kow79]. This is the basis for one of the most relevant properties of LP: declarativeness—i.e. the programmer states *what* the program should do (the logic), while the logic solver decides *how* to do it (the control). Hence, the efficiency of resolution for any given logic program, can be improved without any change in the program itself, simply by changing the solver. Concurrent LP is the field where algorithms and techniques for concurrent/parallel resolution are studied. Informally, it deals whit speeding up computations expressed in LP, by exploiting multi-processor, multi-core, or distributed computing architectures.

Concurrency may be reified into LP in many different flavours. These, in turn, are commonly grouped in two main categories: explicit and implicit—depending on who is in charge of handling concurrency (among the programmer or the solver).

Explicit parallelism. This type of parallelism involves the extension of LP languages with ad-hoc, *explicit* constructs enabling the control of parallelism. This means that the programmer is in charge of controlling concurrency-related aspects manually, through the code.

Explicit control of concurrency needs a highly-skilled programmer who can take advantage of the available constructs to produce very efficient code. However, but it can create difficulties in debugging and testing, and, above all, it may easily break the declarativity of LP—depending on which constructs are provided to the logic programmer, and how much the enable the programmer to decide "what happens when".

There are three sub-types of *explicitly* parallel logic languages:

- those that add explicit *message passing* primitives,
- those that add *blackboard* primitives used by multiple processes running concurrently to communicate with each other (cf. section 15.4),
- those based on *guards*, committed choice, and data flow synchronization.

Implicit parallelism. This type of parallelism involves the capability of the LP engine to perform concurrent operations autonomously, injecting parallelism into logic programs without any programmer intervention. In contrast to the explicit parallelism, there are no extensions to the logic language so that the programmer writes classic logic programs. Instead, concurrency is achieved automatically, by relying on multiple processes/threads/tasks cooperating or competing to explore the proof tree, hence computing solutions to logic queries. Accordingly, the aim of explicit parallelism is to speed up existing and new logic programs without any change to the code.

There are two major sorts of implicit parallelism, namely AND- and OR-parallelism, overviewed below.

AND-parallelism lays in the selection of the next sub-goal to be solved. In particular, it allows the resolution of multiple sub-goals to occur concurrently. This sort of parallelism, as mentioned into [Zha94] and [ZTX93], subtends two relevant situations: *(i)* independent sub-goals, where clauses' literals are independent from each other – as they do not share any variable – and can be concurrently solved, and *(ii)* dependent sub-goals, where clauses' literals are dependent from each other – as they share some variables – and should be solved in some particular order. Of course, the actual complexity of AND-parallelism lays in how dependent sub-goals are handled.

Conversely, OR-parallelism lays in the selection of the clause to be used in the computation of the resolvent. In particular, it allows the concurrent selection of multiple clauses to solve a single (sub-)goal.

Mixing these AND- and OR-parallelism together is of course possible, but it implies an increase of problems that need to be handled.

15.1.2 Why is it interesting

The technology supporting the *sequential* implementation of LP languages has evolved considerably since its birth. In recent years, it has reached a notable state of maturity and efficiency. Today, a wide variety of commercial LP systems and excellent open-source implementations are available that are being used to develop large real-life applications.

For years, LP has been considered well suited for execution on multi-processor architectures. Indeed research in concurrent LP is vast and dates back to the inception of LP itself. Various systems have been developed during the years to support concurrent LP, each one with different features. Unfortunately, most of them are not supported any more, while others are now suffering for the design choices which only made sense for the machines they were designed for.

Since then, technology has evolved moving from machines with limited resources to modern computers with multi-core processors, and lots of memory, improving their performances drastically. Software facilities have evolved hand in hand with the hardware in all areas, like the operating systems, the programming languages, and the concurrency-related primitives they support. This evolution raised the abstraction level of the software technologies available for the mainstream programmer. Hence, nowadays, there exist a plethora of new methods to support concurrent LP, to be explored. Thus, it would be interesting to adapt the vast knowledge on the field of concurrent logic programming to actual technologies, like the Kotlin programming language and its lightweight coroutines, to develop concurrent logic solvers.

15.1.3 Relation w.r.t. the ecosystem

Accordingly, in [Gio21], the authors propose to extend the 2P-Kt ecosystem towards concurrent LP, hence, tilling the soil for future works. In doing so, they formalize and design OR-parallel solvers, as well as their implementation.

More precisely, the authors provide four major contributions. First, they propose a general, object-oriented API supporting the definition and construction of concurrent solvers of any form. Then, they formally describe the design of an OR-concurrent Prolog-like solver via labelled transition systems—following the state-machine approach adopted in chapter 10. Third, they provide an implementation for solvers of such a sort, rooted into the 2P-Kt ecosystem. Last, but not least, they make the concurrent solver compliant w.r.t. the notion of primitive introduced in chapter 10—paving the way towards the creation of concurrent solvers handling stream processing or LP–OOP interoperability scenarios.

The potential of 2P-Kt-powered concurrent LP lays in the possibility of combining concurrency with ILP (cf. section 4.2), as well as the ML-Lib (cf. chapter 12). In all such cases, concurrency may speed up learning without corrupting the declarativeness of LP. In particular, concerning ILP, concurrency may speed up the generation and evaluation of the many possible hypotheses the induction algorithm should step through. Similarly, concerning the ML-Lib, concurrency may enable the simultaneous training of multiple predictors – which, in turn, would speed up any hyper parameters tuning procedure – as well as the simultaneous drawing of multiple predictions via as many predictors—which in turn would speed up the inference phase of any hybrid system.

15.2 Graph Neural Networks for Computational Logic

This section contains contributions from the following works of ours: [ACO21]

Here we discuss the possibility of exploiting graph neural networks (GNN) as a bridge among symbolic and sub-symbolic AI. In particular, we argue that GNN support the sub-symbolic processing of logic knowledge, in all those cases where logic knowledge can be converted into a graph. GNN are a novel topic within

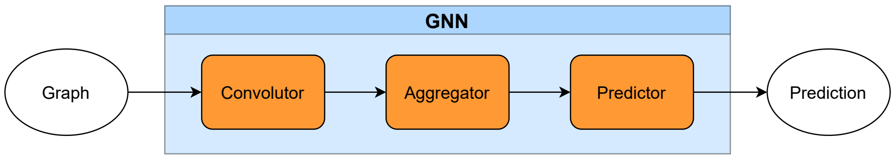


Figure 15.1: Graph Neural Networks are composed as a cascade of simpler blocks

the ML community itself, and, as we further discuss in this section, they have the potential to provide novel solutions for well known CL problems. This may bring several benefits, in particular, in those cases – elicited below – where sub-symbolic processing is expected to perform more efficiently than exact symbolic algorithms.

15.2.1 Brief overview of the field

Most ML approaches can handle data having a *fixed* structure and size—most notably, vectors, matrices, or tensors of real numbers. This may be troublesome in some contexts, given the ever-increasing popularity of applications involving data which cannot be suitably represented by fixed-size, rigid structures. Among the most relevant applications in this category, we can find a number of *graph*-processing scenarios. To tackle this issue, research effort has focused on extending ML approaches to graph-structured data. Notably, graph neural networks [WPC$^+$21] are a novel approach to let ordinary NN-based processing be applied to graphs.

GNN are mathematical models operating upon *directed* graphs, whose vertices (resp., arcs) are labelled with (fixed-size) arrays of real numbers, each one carrying further numeric information about the corresponding vertex (resp., arc). GNN output depends on the learning task to be performed, which commonly ranging in any of three wide classes of tasks: *(i)* the classification of similar graphs having different topology – i.e. *graph classification* – [SK20], *(ii)* the classification of vertices of unknown graphs – i.e. *nodes classification* – [WL20], or *(iii)* the identification of missing but statistically probable arcs—i.e. *link prediction* [FML$^+$19].

Graphs handled by GNN usually carry information in the form of vertices and arcs arrays. Consider for instance the graph representation of a chemical molecule: it is necessary to represent the sort of atomic element associated with each vertex. The same holds for the details of the chemical bonds among two any atoms of a molecule—which must be associated with the graph's arcs.

Figure 15.1 depicts the general architecture of a GNN. It consists of a cascade of three functional blocks (each one composed by one or more layers of neurons) serving specific purposes, namely:

1. the *convolutor* block, which is in charge of accepting the graph G as input and

producing a new *convoluted* graph G' as output, having the same topology of G, where the vector associated with each vertex v has been replaced by another vector describing the relevance of each vertex w.r.t. the whole graph G.

2. the *aggregator* block, which is in charge of computing a fixed-sized, tensorial representation of the graph G' (i.e. an *embedding* of G)
3. the *predictor* block, which acts as an ordinary NN on top of the embedding computed by the previous block, hence supporting regression or classification tasks on the graph G.

Overall, the three blocks constitute a feed-forward NN which can be optimized via gradient descent.

Notably, the convolutor and aggregator blocks aim at converting an arbitrarily-sized graph into a fixed-side tensorial representation, upon with ordinary sub-symbolic processing can be performed. The convolutor block, in particular, relies on convolution operation, extensively exploited in DL to express relevance of local data w.r.t. to global data. However, the application of convolution operation to non-Euclidean data – like graphs – is not straightforward. An equivalent notion of convolution over graphs has been proposed to compute the relevance of each vertex w.r.t. to its neighbours.

15.2.2 Why is it interesting

In [ACO21], we discuss the problem of enabling the sub-symbolic processing of logic knowledge-bases. In particular, we focus on the exploitation of NN as a means to complement CL when it comes to process symbolic data expressed in logic form. More precisely, we study the possible use of graphs as a bridge between CL and neural networks. Notably, we consider graphs as the ideal bridge because of their versatility in representing virtually any sort of data structures, there including recursive ones—an aspect that is very common in logics as well as quite critical in ML.

KB as Graphs. Symbolic knowledge bases can be encoded into graphs in several ways and to serve disparate purposes. Generally speaking, KB can be encoded into graphs by aggregating the graphs attained by encoding all clause therein contained. In all such cases, encoding schemas can act at either the *semantic* or at the *syntactic* level.

Encoding schemas operating at the *syntactic* level capture static relationships inferable from the mere syntax of clauses and KB. Abstract syntax trees (AST) are the simplest example of graphs which can be attained from KB. They consist of

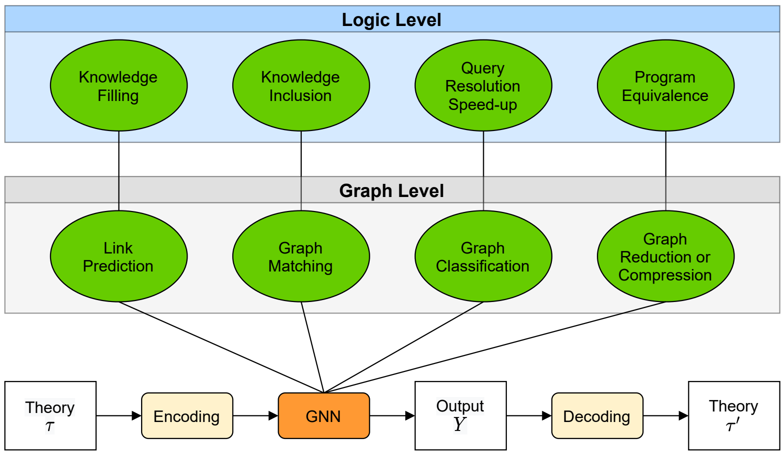


Figure 15.2: General workflow for sub-symbolically processing symbolic knowledge via GNN. Logic level and graph level can be mapped directly, in order for logical problems to benefit from sub-symbolical techniques – e.g. GNN – available at the graph level.

direct acyclic graphs where vertices are of as many sorts as the possible syntactical categories of which may occur in a KB – namely, theories, clauses, predicates, or terms –, whereas arcs simply describe container-contained relations among vertices. Dependency graphs are another kind of graph that may be attained from a KB. They consist of directed graphs where each vertex represents a predicate, and each arc represents a logic dependency among two predicates—meaning that the predicate corresponding to the destination vertex must be proven true before the predicate corresponding to the source vertex, in a resolution process.

Encoding schemas operating at the *semantic* level capture high level relationships that can be inferred from the actual meaning of a logic theory. Entity-Relationship (ER) graphs are the simplest kind of graph in this category. They aim at expressing via graphs the same information a ground KB expresses via formulæ. They consist of directed graphs where vertices may either represent entities (i.e. terms) or relationships (i.e. predicates) and arcs represent the participation of an entity into a relationship. Triplet graphs are another simple way of representing ground theories where all terms are constants and all predicates are either unary or binary. When this is the case, each constant is considered an entity, binary predicates are considered as relations among two entities, and unary predicates are considered as properties an entity may or may not have. Thus, a graph can be attained by defining a vertex for each different constant in a KB, and arc for each couple of constants involved in at least a binary predicate.

Handling logic tasks via GNN. Manipulation of logic knowledge enables the resolution of complex queries via logical inference. There exist, however, relevant tasks which are hard to formalise or solve into the logic realm, because of either their numerical nature or algorithmic infeasibility. Here, in particular, we identify four relevant operations on knowledge bases for which, we argue, it is worth investigating sub-symbolic solutions.

The tasks considered – shown in the upper box of fig. 15.2 – are *(i)* knowledge filling, *(ii)* knowledge inclusion, *(iii)* program equivalence, and *(iv)* resolution speed-up.

Knowledge Filling. Entities and relations available in a logic theory may sometime lack some instances. For example, this may happen because the human operator handcrafting the theory was imprecise or when an agent's knowledge is incomplete. When this is the case, we consider the knowledge base as *fragmented.*

To deal with such fragmented theories, it may be useful to identify missing relations between existing entities. This task may be tackled via statistical analysis of the theory under examination, which may lead to the identification of latent relations among entities.

A knowledge filling problem would be hard to handle symbolically, as logic reasoners commonly struggle in processing knowledge they do not have. While most solvers operate under a closed world assumption – letting them considers as false everything they do not explicitly know to be true –, even the ones operating under an open world assumption do not commonly include mechanisms to generate new knowledge out of thin air. In all such cases, the coherence and completeness of the knowledge base is usually considered as an a-priori requirement for logic computations to work properly. Conversely, in the sub-symbolic realm, semantic similarities among the entities and relations of a logic theory may be better captured, which may help reconstructing missing facts.

Consider for instance the case of a simple theory representing kinship relationships. The lack of a single relation – say that "John and Mary are siblings" – may significantly hinder a solver's ability to deduce kinships among entities of a family—e.g. "the sons of John and Mary are cousins". The solution for this task is not straightforward, thus attracting our attention.

Knowledge Inclusion. Knowledge inclusion represents the task checking whether a given theory (usually smaller) is complementary w.r.t. another given theory (usually larger) or not. The same clauses could occur with slightly-different shapes—e.g. using different predicates/functor names or different positions arguments in the same predicates.

This task requires the ability to express equivalence or similarity among *groups* of clauses, which is not straightforward [EFTW04, JWHY15]. Computing exact solutions to this problem may soon become infeasible as the dimensions of the involved theories increases. Conversely, in the sub-symbolic realm, the same problem may be modelled as a pattern-matching problem. This may pave the way towards the computation of *approximate* solutions to the knowledge inclusion problem in reasonable time.

As an example, consider multiple agents sharing partially-similar information. In this case, it would be desirable to identify agents common knowledge and ease their interaction. Suppose that agents information is expressed via two theories τ_1 and τ_2, both representing family trees. τ_1 expresses 1^{st} degree relatives only, while τ_2 includes also 2^{nd} degree relatives. τ_1 may consider more/less/different family members w.r.t. τ_2, and kinships may also be defined in different ways between the two theories. However, τ_1 is – logically speaking – a subset of τ_2, and we need to detect this property.

Program Equivalence. Program equivalence represents the task of computing a simpler and equivalent theory τ' starting from a theory τ. This may imply removing redundancies and simplifying clauses. As for the knowledge inclusion task, program equivalence requires a procedure to compare sets of clauses, other than the capability of generating reduced equivalent variants of clauses. It is our opinion that both these procedures may be better expressed into sub-symbolical realm.

Considering again agents storing kinships information, it may be desirable to compress a single agent information to produce a new theory for a simpler agent. This new theory should ideally have fewer rules, while spanning the same family tree of the original theory. Such a requirement is difficult to satisfy, and would probably require notions of semantically-equivalent sets of kinships—e.g., the set of relations containing only *parent* spans the same family tree of the set of relations $\{mother, father\}$.

Query Resolution Speed-Up. Logic theories are commonly exploited by logic solvers to draw inferences, via some resolution procedure. The execution time of any query resolution vastly depends on the complexity of the algorithm(s) expressed by the logic theory. To this regard, a number of efficiency tweaks may affect the execution time in the average case. For instance, the solutions to most frequent queries may be cached, or smart strategies may be employed to affect the way the solver explores a solution space. However, caching costs space, whereas any *rigid* resolution strategy may result efficient on some sorts of queries, while still being slow on some others.

In all those cases, sub-symbolic sub-systems capable to learn from experience can bring about huge benefits. There, a sub-symbolic helper may be trained to predict the outcomes of most frequent queries, thus speeding up queries with constant space requirements. Furthermore, an online learning procedure may be injected into the solver, making it adapt the resolution strategy to the query at hand, on the basis of the experience accumulated via previous queries.

This task may be particularly relevant when considering real-time agents working with complex knowledge bases. Focusing again on kinships, when the number of family members is huge and relations between family members are complex – e.g., fourth grade cousins –, query resolution may suffer from delays hindering agents ability to make real-time decision and perform real-time tasks. Therefore, it may be interesting to use techniques that aim at speeding up the resolution of queries over such huge theories. Sub-symbolical approaches may ease this task, by compressing theory knowledge to simple and easy-to-handle embeddings.

Graphs as Bridges. Here we discuss the role of GNN in addressing the relevant logic tasks from the paragraph above. In particular, we show how all such tasks can be mapped onto known graph-related problems which can be addressed via GNN. In other words, we comment the upper part of fig. 15.2.

Knowledge filling → Link prediction. The knowledge filling task usually requires semantic knowledge to be taken into consideration. Therefore, to map the knowledge filling task to an equivalent problem over graphs we should consider preserving the semantic information of the theory. We can then assume to map entities of a theory to vertices of a graph. Rules and relations can then be represented as vectorised arcs existing between the graph vertices. Each position of an arc vector represents a specific relation, preserving the original semantic of the theory. In this scenario, the task of predicting possible missing relations or rules is mapped to the problem of identifying which arcs are missing from the graph.

Knowledge Inclusion → Graph matching. In the same way as for the knowledge filling task, knowledge inclusion requires semantic knowledge of the theory to be taken into account. Therefore, we require the mapping between logic and graphs to preserve the theory semantic. Moreover, knowledge inclusion requires a comparison between two or more theories: entities and relations from a theory should be compared to their counterparts of the other theory and matched upon need.

As done for knowledge filling, let us assume entities to be represented as vertices, and rules or relations as vectorised arcs. The mapping produces as many graphs as the theories available for the inclusion task. Therefore, from a graph perspective, knowledge inclusion is mapped to a graph matching problem. Indeed, the two or more graphs corresponding to their theory counterparts should be matched for some portion of them.

The matching between graphs is still an open research problem, as it is computationally very expensive, but is easier to tackle than rules and entities matching. This holds in particular whenever entities do not match exactly, or, rules share analogous semantics but are defined in different forms—e.g., parent and mother.

Program equivalence → Graph compression. Given a specific theory, program equivalence aims at obtaining a simpler – smaller – theory that preserve the same expressiveness. Depending on the considered approach the mapping between logic and graph level may bear different requirements. In its simplest form program equivalence requires to simply remove unnecessary relations and rules of a theory to compress it. This approach does not require explicitly the semantic level to be considered while processing the theory. More interestingly, program equivalence may also require to map set of rules and relations to a single (or a smaller set of) rules(s). This increased complexity introduces the need for semantic to be taken into account and to be preserved in the mapping from logic to graphs. If we consider the same mapping of previous examples, program equivalence can be linked to the graph compression problem. Indeed, obtaining a smaller set of equivalent rules and entities can be done removing or merging together arcs and vertices of the graph theory counterpart.

Query resolution speed-up → Graph classification. Query resolution speed-up aims at obtaining faster execution of given queries over a logic theory. It may be helpful for query resolution to maintain the semantic information embedded in the theory. Therefore, the mapping between logic and graphs may benefit from the preservation of semantic information, and generally speaking vastly depends on the requirements of the desired speed-up. Differently from previous tasks, for query resolution speed-up we consider obtaining graphs for queries to be solved—rather than a single graph for the whole theory. The graph representing a query is matched with the query resolution, considered as the graph label. Following this mapping, the query resolution speed-up is mapped to a graph classification problem, where the label of a graph should be predicted. Any approach can then be leveraged to classify graphs—i.e. obtain query solutions. This approach may not be significant for simple queries applied to small knowledge basis. However, it may result in great speed-up when complex knowledge bases and queries are considered. Indeed, GNN scalability over large graphs is mostly not an issue, resulting in quick graph classification.

15.2.3 Relation w.r.t. the ecosystem

Despite the 2P-Kt ecosystem is not explicitly mentioned in [ACO21], its potential role in this framework is straightforward. In fact, as clearly shown in fig. 15.2, any GNN-powered workflow aimed at processing symbolic information shall accept logic theories as inputs, and, possibly, produce logic theories as output. In both cases, the existence of a software library supporting the representation and manipulation of clauses and theories is of paramount importance. Despite this may seem a technical marginal aspect, it is indeed a key enabling factor for any pre- and post-processing algorithm to be applied before or after the sub-symbolic computation is triggered.

15.3 Symbolic Knowledge Injection

This section contains contributions from the following works of ours: [MCO22]

While symbolic knowledge extraction (cf. chapter 6, and chapter 13) is the preferred way to provide explanations within the scope of this thesis, here we provide insights about a dual approach, namely symbolic knowledge injection (SKI). SKI is not exactly a keyword from the literature, but rather an umbrella term we use to denote a plethora of methods from the recent literature of knowledge graph embedding, and neuro-symbolic computation. Intuitively, it deals with putting symbolic knowledge into sub-symbolic predictors. In this section, we provide a brief overview of what we mean by SKI and how it may be useful within the ML playground to work around the lack of interpretability.

15.3.1 Brief overview of the field

We denote as “symbolic knowledge injection” the task of letting a sub-symbolic predictor exploit formal, symbolic information to improve its predictive performance (e.g. accuracy, f1-measure, learning time) over data. Unlike numeric data upon which predictors are commonly trained, symbolic data is generally more compact and expressive, as intensional representations of complex concepts may be concisely written. In particular, symbolic information may encode bold rules that must be satisfied by the concepts the predictor is willing to learn. Hence, provided that some SKI procedure is available, data scientists may craft ad-hoc collections of symbolic expressions aimed at aiding the training of a particular predictor, for a specific learning task. In other words, injection enables provisioning prior knowledge – namely, the designer’s common sense – to ML predictors under training.

When it comes to neural networks, approaches for SKI are manifold, and the literature on this topic is vast. Broadly speaking, there exist three major sorts of approaches supporting the injection of symbolic knowledge into neural networks. In particular, SKI can be performed by *(i)* constraining, *(ii)* structuring, or *(iii)* feeding the network via symbolic knowledge.

Approaches of the first sort perform injection during the network's training, using the symbolic knowledge as either a constraint or a guide for the optimization process. The works by [DGS17, MGDG19] describe methods of this sort. Conversely, approaches of the second sort perform injection by altering the network's architecture to make it mirror the symbolic knowledge. Works by [Bal86, THA92] describe methods of the latter sort. Finally, approaches of the third sort attempt to *embed* symbolic knowledge into tensorial form to then be processed via ordinary NN, as well as other ML predictors. The GNN-based manipulation of logic theories discussed in section 15.2 falls under this category, as well as the many methods for knowledge graph embedding—nicely surveyed in [WMWG17].

Accordingly, all such approaches may leverage upon one or more strategies to manipulate symbolic knowledge. One strategy works by generating structured data form the symbolic information to use it in conjunction with the training set (see [vRMG$^+$19, ZYH$^+$18]) to train the predictor. Another strategy works by encoding the symbolic knowledge via fuzzy logic, i.e. into functions of real numbers. Such functions could then be exploited to interpret symbolic knowledge numerically, e.g. as constraints for the optimization process subtended by training (cf. [DRG17, DGS17, MGDG19]). One further strategy may be exploited where convolution-like operations (such as GNN) are exploited to distil concise, fixed-size representations of the symbolic information to later feed the sub-symbolic predictor with.

Concerning the symbolic knowledge, virtually all techniques we are aware of require information to be represented via (some subset of) first order logic (FOL) formulæ. Actual methods may then vary, depending on *(i)* which particular sub-set of FOL they rely upon, *(ii)* how are logic formulæ interpreted as constraints, and *(iii)* whether formulæ require to be *grounded* or not, before SKI can occur. To the best of our understanding, most of the methods proposed so far in th literature require the knowledge base be ground, at some point in the SKI process. When this is not the case, the KB should be grounded—which of course is only possible if the Herbrand base of the KB is finite. This, in turns, subtends hidden methodological constraints for the current state of the art of SKI, which commonly only supports *strict* sub-sets of FOL—where, e.g. structure symbols are forbidden.

15.3.2 Why is it interesting

As discussed in chapter 6, a major issue in supervised ML is the usage of *opaque* predictors – such as NN – as black boxes. It is not trivial to understand what NN actually learn from training data, and how they generalise from that data to the whole domain. Currently, state of the art solutions address this issue by supporting their inspection via a plethora of different mechanisms [GMR+19]. On our side, the same issue is tackled via symbolic knowledge extraction, as discussed in chapter 13.

However, a third way is possible. Thanks to SKI, sub-symbolic predictors can be made safe *by-construction*, as they can be trained by taking the the designer's common sense – suitably represented in symbols – into account. In other words, SKI lets us circumvent the opacity issue of ML predictors. Indeed, the injected knowledge may encourage the NN to learn desired information, while preventing it from violating the designer's constraints—expressed by symbols.

More generally, SKI enables a higher degree of *control* over a sub-symbolic predictor and its behaviour, constraining it with human-like common-sense—suitably encoded into symbolic form. In this way, data scientists may *correct* a misbehaving predictor or train one despite lack of examples describing a particular phenomenon. For example, symbolic knowledge may encode impossible or meaningless situations to be avoided (e.g. "if X has two legs, then X cannot be a cat"), as well as obvious (for the human) relations which may be hard (for the predictor) to learn from data alone (e.g. "if X is Y's parent, then Y is X's child").

Furthermore, the greatest potential comes from the *combined* exploitation of SKE and SKI. Indeed, they can be exploited cascade-like fashion where sub-symbolic predictors are *(i)* inspected (via SKE), *(ii)* debugged, and *(iii)* fixed (via SKI) by data scientists, as part of their ML workflows.

15.3.3 Relation w.r.t. the ecosystem

SKI is dual w.r.t. to SKE, as the former aims at injecting symbolic knowledge into ML predictors, and the latter aims at extracting it.

In chapter 13, we propose the design and implementation of PSyKE, i.e. our platform for SKE. There, the 2P-Kt ecosystem supports the construction of logic rules, as part of the extraction process.

We envision a similar, yet dual platform for SKI, namely PSyKI, providing a general-purpose notion of *injector*, with many possible implementations—each one reifying a particular injection algorithm/method. There, the 2P-Kt ecosystem would support the representation and (programmatic) manipulation of the symbolic knowledge acting as input of the injection process.

15.4 Tuple-based Coordination

This section contains contributions from the following works of ours: [CROM19]

This section contains contributions from the following Bachelor's thesis: [Riz18], which we supervised

Here we discuss the role of the 2P-KT logic ecosystem to support tuple-based coordination, i.e. a well-established research field studying models, mechanisms, and technologies aimed at governing the interaction of intelligent agents in concurrent and distributed systems. There, the logic plays a fundamental role in the representation and manipulation of data and events produced/consumed by agents during interaction. In particular, we report the proposal of TuSoW, a tuple-based technology leveraging on 2P-KT to support coordination among distributed agents interacting via most common network protocols.

15.4.1 Brief overview of the field

Coordination is the discipline of enabling and constraining interactions among software components [Cia96] or, more generally, of managing the dependencies among activities [MC94]. In a world were billions of computational machines have access to the Internet, this translates to governing the interactions among devices and services, so as to fully realise the system functionality, through appropriate technologies backed by well-founded models.

Among the many coordination models, tuple-based ones are the most studied [CMO$^+$18], mainly due to their expressiveness, elegance, and flexibility. There, interaction among computational entities of a system are mediated by a *tuple space*, the data repository ruling *associative access* to information chunks called *tuples*. Tuples may represent messages by interacting components, data they intend to share, or a reification of the events of interest for them. In tuple spaces, interaction is governed by defining *how* and *when* agents are allowed to *insert*, *read*, or *consume* data.

Linda is the archetypal tuple-based coordination model [Gel85]. It lets interacting agents share tuples in tuple spaces by means of three primitive operations: `out`, to produce a tuple; `rd`, to read a tuple; and `in`, to consume a tuple. The set of the basic Linda primitives could be interpreted as constituting a sort of API of a tuple space, whose essential features enable agents to synchronise their activities (hence coordinate): *generative* communication, *associative* access, and *suspensive* semantics.

Generative communication makes tuples exist in tuple spaces *independently* of their creators – that is, tuples sprout with an `out` and die with an `in`, regardless of the life of the producer –, and supports *reference* (or, *name*), *time*, and *space uncoupling* among interacting processes. Associative access makes tuples be accessed (i.e. either consumed or observed) through *templates* predicating over their content – e.g., regular expressions can work as templates for tuples represented as strings –, and supports dealing with *partial information* and *incomplete knowledge*: *getter primitives* (`rd`, `in`) expect a template as their argument and return a *matching tuple*. Finally, suspensive semantics makes getter primitives *wait to be served* until a tuple matching given the provided template becomes available – when the waiting primitive is eventually *completed* returning the matching tuple –, and supports *synchronisation* (ordering) of actions, hence coordination of activities.

15.4.2 Why is it interesting

As witnessed by the large amount of LINDA-based technologies exploiting FOL terms as tuples (cf. [CDMSL+20]), the history of tuple-based coordination is deeply entangled with CL. There, FOL brought key benefits to the field of Coordination, and vice versa.

Indeed, from a Coordination perspective, FOL enables *(i)* the exchange of arbitrarily complex data, possibly intensionally represented, and *(ii)* powerful and expressive associative access, based on logic unification. Furthermore and foremost, FOL enables the coordination of intelligent software agents via the *direct* exchange of logic knowledge. This particular aspect is considered very relevant within the MAS community.

Conversely, from a logic perspective, LINDA's tuple spaces provide a means to support explicit parallelism among two or more logic solvers. There, logic solvers act as independent agents running in parallel, and tuple spaces act as shared blackboards upon which all such agents can read, write, or delete tuples to coordinate their reasoning processes. This is in turn the basic brick enabling many sorts of competitive or cooperative protocols.

15.4.3 Relation w.r.t. the ecosystem

Unfortunately, logic-based technologies in the field of Coordination are characterised by an high degree of obsolescence, as we further discuss in [CDMSL+20]. In many cases, technologies are unmaintained, proof of concepts, or based on other logic-based technologies subject to the same issues. The overall effect is that most of them are inadequate to serve the needs of modern Web-, Cloud-, or Edge-based applications, because of their poor interoperability. To mitigate this issue, in [CROM19], we propose a layered reference model, architecture, and technology for tuple-based coordination, namely TUSOW (Tuple Spaces over the Web).

TUSOW is modelled according to the *Coordination as a Service* (Coord-aaS) architecture [VO06], where clients are the software entities requiring coordination services to achieve their goals. Clients may communicate with services, hence

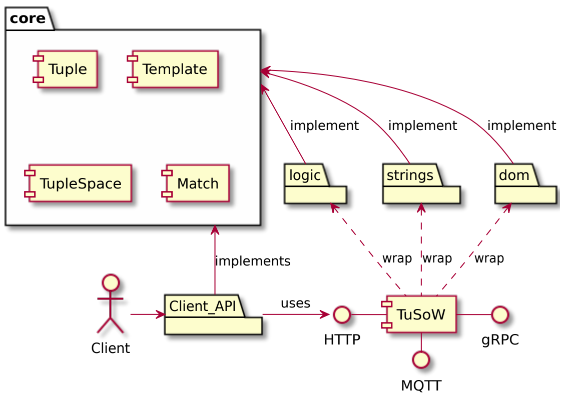


Figure 15.3: TuSoW architecture. Circles represent TuSoW remote API. Clients API exploiting MQTT and gRPC remote API are not depicted.

interact *through* tuple spaces, by means of a set of *remote API* geared toward different settings, including HTTP, WebSockets, gRPC, and MQTT. This is made possible by TuSoW's modular architecture, whose main components are shown in fig. 15.3.

At the architectural level, TuSoW consists of a core module defining the base types for tuples, templates, matchings, and tuple spaces, which is then specialised by different implementation modules—one for each pair of tuple-template languages. All implementations support *local* coordination of multiple concurrent processes, threads, or agents running on the same machine. Distribution is enabled by the *remote API*. Remote API reifies the Coord-aaS notion upon different technologies, targeting different settings. Currently, TuSoW supports three mainstream technologies widely used in CPS and WoT/IoT scenarios: HTTP, gRPC, and MQTT.

Notably, TuSoW represents data (e.g. tuples) and queries (e.g. templates) is modular as well, and independent of the remote API. The idea of making Linda primitives orthogonal w.r.t. the specific tuple / template language adopted is not new, as it has already been extensively discussed in [Omi99], for instance. Nevertheless, to the best of our knowledge, no existing implementation allows for this, although in [MOC17] an intuition concerning the potential of supporting several tuple / template languages is discussed.

In particular, TuSoW lets clients represent tuples with

- YAML, JSON, and XML formats — to target web-related contexts such as SOA and RESTful web services;
- first order logic (FOL) terms – in particular Prolog concrete syntax – to target intelligent agents and multi-agents systems;

Listing 15.1: Inducer interface, tailored onto the 2P-Kt API

```
interface Inducer {
    fun induce(
        positiveExamples: Iterable<Clause>,
        negativeExamples: Iterable<Clause>,
        background: Theory
    ): Theory
}
```

- plain text documents as a fallback for contexts where structuring knowledge is neither possible nor desirable.

In principle, any other data representation format / language may be supported, thanks to TuSoW modular architecture. Each data representation format comes with one or more preferred ways for retrieving or querying data stored with that format, hence, to express templates and perform matching. For instance, JSON or YAML data can be queried through JsonPath, in the same way as XML data are queried through XPath. Plain text is often queried through Regular Expressions, whereas FOL terms may be matched against other FOL terms through unification. Of course, the exploitation of terms and unification in TuSoW is made possible by 2P-Kt.

15.5 Inductive Logic Programming

This section contains contributions from the following Master's thesis: [Spe21], which we supervised

The field of ILP is extensively presented in section 4.2, as well as its role within the scope of this thesis. Here, we simply discuss the (potential) relation among ILP and the 2P-Kt ecosystem. In doing so, we report the discussion proposed in [Spe21], where the design of a 2P-Kt module for ILP is sketched.

There, the authors propose a general-purpose, object-oriented API for ILP, namely, via the `Inducer` interface, represented in listing 15.1. Essentially, an *inducer* is any object capable of performing logic induction on a number of positive and negative examples (which are clauses, in the general case), and a background theory. The result of any induction process, in the general case, is theory as well—possibly containing one or more clauses.

Virtually all ILP algorithms may be wrapped behind this general interface. In this sense, the main contribution of [Spe21] is to provide general support for ILP, at the OO level. This, in turn, is very interesting as it represents a first step towards the provisioning of a principled library supporting the engineering of ILP solutions, where programming-in-the-large tools may be exploited, and ILP itself is not bound to any particular LP language.

ILP algorithms may be added to the ecosystem by simply writing new implementations for the `Inducer` interface. These implementations may then be exploited in main-stream programming as well as in any logic solver built on top of the 2P-KT ecosystem.

Epilogue

Chapter 16

Conclusions

In this chapter, conclusions are drawn and the achievement of the goals of this thesis is discussed. Instead, future research directions are discussed in chapter 15.

As outlined in chapter 1, this thesis tackles the problem of bridging computational logic and data science, as a particular case of the more general problem of combining symbolic and sub-symbolic AI. Along this line, our contribution is split in two major parts.

The computational perspective. In part I, we analyse CL and DS w.r.t. four major dimensions, under a computational perspective. Hence, stemming from an historical perspective on the two branches of AI, we draw a detailed comparison about how knowledge is represented, learnt, inferred, and explained both in symbolic and sub-symbolic AI. Coherently with the goals 1 and 2 (cf. chapter 1), we identify the key differences and complementarities among the two, other than the most relevant issues arising when *hybrid* systems – i.e. systems jointly exploiting CL and DS – are designed.

Key differences. Among the key differences, it is worth mentioning the way knowledge is represented, and learnt.

Concerning representation (cf. chapter 3), the main difference lays in the ability to represent information intensionally—i.e. implicitly, as opposed to the need of extensionally describing each single datum. Indeed, this is, at the time of writing, is a prerogative of CL, whereas DS most often requires knowledge to be provided in the form of numeric arrays.

Consequently, as far as learning is concerned (cf. chapter 4), another key difference lays in the way useful knowledge is acquired from novel evidence—i.e. data. While DS commonly relies on numeric algorithms to fit the parameters of a target function, CL relies upon symbolic algorithms which are capable of learning full logic relations.

Furthermore, one may empirically observe how numeric algorithms are inherently data-eager, while symbolic ones can attain good learning performances even in presence of very few examples. However, in practice, it is worth remarking how the two learning approaches are heavily unbalanced when it comes to technological support. Indeed, while technologies for numeric learning are flourishing, the same is not true for symbolic learning.

Key complementarities. Among the key complementarities, it is worth mentioning the way knowledge relates to both inference, and explanation.

Concerning inference (cf. chapter 5), we stress that CL and DS subtend different – yet complementary – ways of interpreting inference. Indeed, CL subtends a rational way of drawing conclusions out of premises – i.e. reasoning, in a nutshell –, possibly following a deductive, abductive, or inductive strategy. Conversely, DS subtends an intuitive way of recognising patterns in (possibly, unseen) data, hence interpreting inference as a form of perception. Notably, the two ways are complementary rather than competing. Hybrid systems may be designed where fuzzy tasks are devoted to sub-symbolic processing, whereas higher-level, crisp, decision-taking tasks are devoted to some symbolic component.

As far as explanation is concerned (cf. chapter 6), we acknowledge that the interpretation of symbols is straightforward in CL, while it may easily become cumbersome when knowledge is represented through arrays of numbers—as in DS. Accordingly, explainability of ML predictors makes the complementarity among symbolic and sub-symbolic AI even more evident. Along this line, we model the explanation the act of extracting symbolic knowledge out of sub-symbolic predictors—while of course guaranteeing the extracted knowledge actually reflects what the predictors has learnt. Notably, this is a relevant contribution within the field of XAI, which is currently striving to make ML predictors explainable.

The technological perspective. In part II, we analyse the state of the art of CL under a technological perspective.

Coherently with goal 3, we assess the currently available logic-based technologies w.r.t. their capability to serve the needs of modern and future AI (cf. chapters 7 and 8)—possibly, in a synergy with DS. It turns out a considerable amount of logic-based technologies is nowadays unmaintained, while many others are flourishing within the MAS community (cf. chapter 8). However, we observe a tendency towards the creations of what we call "technological silos", i.e. poorly interoperable technologies serving specific purposes—despite very well. Interoperability issues arise for a number of reasons, mostly related to design or technological choices (e.g. the runtime platform), which made sense sense in the past but are constraining nowadays.

Accordingly, we address goal 4 by designing a notion of logic ecosystem, and by reifying into the 2P-Kt technology, as discussed in chapter 9. There, a logic ecosystem consists of a collection of loosely coupled software modules, each one supporting a particular CL aspect, notion, or functionality in an unopinionated way. In other words, it is designed in such a way to support CL – as well as its interoperability with DS – without committing to any particular functionality of use case. Ad-hoc modules are designed for knowledge representation and automated reasoning, as basic functionalities, while the addition of further modules targetting, e.g., learning or explanation is enabled by building on top of the base modules and their open API. Furthermore, to prevent 2P-Kt from becoming a technological silos itself, we have designed as a *multi-platform* software ecosystem.

Consequently, in the subsequent chapters, we address goal 5 by designing – and possibly implementing – a number of extensions for our logic ecosystem pushing it towards DS. Notably, such extensions serve the purpose of demonstrating 2P-Kt's interoperability as well. In particular, we develop extensions bridging DS and our logic ecosystem in several ways. For instance, we bridge the ecosystem with data stream processing (chapter 10), mainstream programming paradigms such as object-oriented and functional programming (chapter 11), machine learning (chapter 12), eXplanable AI (chapter 13), and probabilistic logic programming (chapter 14). A number of further bridges are envisioned and discussed in chapter 15, concerning for instance concurrent logic programming, graph neural networks, symbolic knowledge injection, tuple-based coordination, and inductive logic programming.

Hopefully, in the future, our logic ecosystem will be enriched enough to act as the most adequate conceptual and technological basis for hybrid intelligent systems.

Bibliography

[IS95] ISO/IEC JTC 1/SC 22 Technical Committee. ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core. International Standard ISO/IEC 13211-1, ISO/IEC, 1995.

[2P-21] 2P-Kt. Home page. `https://github.com/tuProlog/2p-kt`, 2021.

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Abi08] Syed Sibte Raza Abidi. Healthcare knowledge management: The art of the possible. In David Riaño, editor, *Knowledge Management for Health Care Procedures*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[ACO21] Andrea Agiollo, Giovanni Ciatto, and Andrea Omicini. Graph neural networks as the copula mundi between logic and machine learning: A roadmap. In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *WOA 2021 – 22nd Workshop "From Objects to Agents"*, volume 2963 of *CEUR Workshop Proceedings*, pages 98–115, Bologna, Italy, October 2021. Sun SITE Central Europe, RWTH Aachen University.

22nd Workshop “From Objects to Agents” (WOA 2021), Bologna, Italy, 1–3 September 2021. Proceedings.

[Ada21] Amina Adadi. A survey on data-efficient algorithms in big data era. *J. Big Data*, 8(1):1–54, 2021.

[ADBGDP04] Veronique Adriaenssens, Bernard De Baets, Peter L. M. Goethals, and Niels De Pauw. Fuzzy rule-based models for decision support in ecosystem management. *Science of the Total Environment*, 319(1–3):1–12, 2004.

[ADT95] Robert Andrews, Joachim Diederich, and Alan B. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowl. Based Syst.*, 8(6):373–389, 1995.

[AES17] Kazi Masudul Alam and Abdulmotaleb El Saddik. C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems. *IEEE Access*, 5:2050–2062, 2017.

[AFR$^+$10] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2010.

[AFWZ02] Alessandro Artale, Enrico Franconi, Frank Wolter, and Michael Zakharyaschev. A temporal description logic for reasoning over conceptual schemas and queries. In *European Workshop on Logics in Artificial Intelligence (JELIA 2002)*, pages 98–110. Springer, 2002.

[AH13] Khalid Almohammadi and Hani Hagras. An adaptive fuzzy logic based system for improved knowledge delivery within intelligent E-Learning platforms. In *2013 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2013)*, pages 1–8. IEEE, 2013.

[AK12] M. Gethsiyal Augasta and T. Kathirvalavakumar. Reverse engineering the neural networks for rule extraction in classification problems. *Neural Process. Lett.*, 35(2):131–150, 2012.

[AKD$^+$10] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010.

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 27(6):509–516, 1978.

[AL03] John R. Anderson and Christian Lebiere. The newell test for a theory of cognition. *Behavioral and Brain Sciences*, 26(5):587–601, 2003.

[ALS12] Arnulfo Azcarraga, Michael David Liu, and Rudy Setiono. Keyword extraction using backpropagation neural networks and rule extraction. In *The 2012 international joint conference on neural networks (IJCNN)*, pages 1–7. IEEE, 2012.

[Ama21] Amazon.com, Inc. Djl - deep java library. `https://djl.ai`, 2021.

[AMPV06] Reza Akbarinia, Vidal Martins, Esther Pacitti, and Patrick Valduriez. Design and implementation of atlas p2p architecture. *Global data management*, 8:98, 2006.

[APG$^+$11] Arash Azadegan, Lejla Porobic, Sepehr Ghazinoory, Parvaneh Samouei, and Amir Saman Kheirkhah. Fuzzy logic in manufacturing: A review of literature and a specialized application. *International Journal of Production Economics*, 132(2):258–270, 2011.

[Apt90] Krzysztof R Apt. Logic programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990:493–574, 1990.

[Apt01] Krzysztof R. Apt. *The Logic Programming Paradigm and Prolog*, chapter 15, pages 475–508. Cambridge University Press, 2001.

[ARAA$^+$16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien

Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[ARFS12] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Real-time complex event recognition and reasoning–a logic programming approach. *Applied Artifical Intelligence*, 26(1-2):6–57, 2012.

[AvKLM01] Maysam F. Abbod, Diedrich G. von Keyserlingk, Derek A. Linkens, and Mahdi Mahfouf. Survey of utilisation of fuzzy technology in medicine and healthcare. *Fuzzy Sets and Systems*, 120(2):331–349, 2001.

[Baa03] Franz Baader. Basic description logics. In *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95, USA, 2003. Cambridge University Press.

[Bal86] Dana H. Ballard. Parallel logical inference and energy minimization. In Tom Kehler, editor, *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August*

11-15, 1986. Volume 1: Science, pages 203–209. Morgan Kaufmann, 1986.

[BB10] Nahla Barakat and Andrew P. Bradley. Rule extraction from support vector machines: A review. *Neurocomputing*, 74(1):178–190, 2010. Artificial Brains.

[BBCP] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. JADE — a Java agent development framework. chapter 5, pages 125–147.

[BBD+06] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, A. El F. Seghrouchni, Jorge J. Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, January 2006. Special Issue "Hot Topics in European Agent Research II".

[BBHG17] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *Journal of Machine Learning Research*, 18:109:1–109:67, 2017.

[BC02] Roberto Bagnara and Manuel Carro. Foreign language interfaces for Prolog: A terse survey. *ALP Newsletter*, 15(2), May 2002.

[BC04] Tijl D. Bie and Nello Cristianini. Convex methods for transduction. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in neural information processing systems*, pages 73–80, 2004.

[BCMP20] Federico Bergenti, Giovanni Caire, Stefania Monica, and Agostino Poggi. The first twenty years of agent-based software development with JADE. *Autonomous Agents and Multi Agent Systems*, 34(2):36, 2020.

[BD08] Nahla Barakat and Joachim Diederich. Eclectic rule-extraction from support vector machines. *International Journal of Computer and Information Engineering*, 2(5):1672–1675, 2008.

[BDKT97] Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial intelligence*, 93(1–2):63–101, 1997.

[BDTE18] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Lars: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence*, 261:16–70, 2018.

[BEF17] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, 2017.

[Ber08] Merrie Bergmann. *An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems.* Cambridge University Press, 2008.

[BFOS84] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees.* Chapman & Hall/CRC, 1984.

[BG11] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[BH06] Rafael H. Bordini and Jomi F. Hübner. BDI agent programming in AgentSpeak using Jason. In Francesca Toni and Paolo Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006.

[Bis01] Stefano Bistarelli. *Soft Constraint Solving and programming: a general framework.* PhD thesis, Computer Science Department, University of Pisa, 2001.

[BKC94] G. Baues, P. Kay, and P. Charlier. Constraint based resource allocation for airline crew management. *Proc. ATTIS*, 94, 1994.

[BL04] Ronald J. Brachman and Hector J. Levesque. The tradeoff between expressiveness and tractability. In Ronald J. Brachman and Hector J. Levesque, editors, *Knowledge Representation and Reasoning*, The Morgan Kaufmann Series in Artificial Intelligence, pages 327–348. Morgan Kaufmann, San Francisco, 2004.

[Bla08] S. Blackburn. *The Oxford Dictionary of Philosophy.* Oxford Paperback Reference. OUP Oxford, 2008.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[BM88] Robert S. Boyer and J. Strother Moore. *A computational logic*. Academic Press Professional, Inc., USA, 1988.

[Boj07] George Bojadziev. *Fuzzy logic for business, finance, and management*, volume 23. World Scientific, 2007.

[Bol00] Guido Bologna. A study on rule extraction from neural networks applied to medical databases. In *The 4th European Conference on Principles and Practice of Knowledge Discovery (PKDD2000), Lyon, France*, pages 1–11, 2000.

[Bou04] Marc Boullé. Khiops: A statistical discretization method of continuous attributes. *Machine Learning*, 55(1):53–69, 2004.

[BP97] Guido Bologna and Christian Pellegrini. Three medical examples in neural network rule extraction. *Physica Medica*, 13:183–187, 1997.

[BPr21] BProlog. Home page. `http://www.picat-lang.org/bprolog`, 2021. Last access: April 17, 2022.

[BR13] Elena Bellodi and Fabrizio Riguzzi. Expectation maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2):343–363, 2013.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–533. North Holland, 2001.

[BSDL$^+$01] Bart Baesens, Rudy Setiono, V. De Lille, Stijn Viaene, and Jan Vanthienen. Building credit-risk evaluation expert systems using neural network rule extraction and decision tables. In Veda C. Storey, Sumit Sarkar, and Janice I. DeGross, editors, *Proceedings of the International Conference on Information Systems, ICIS 2001*, volume 20, pages 159–168. Association for Information Systems, 2001.

[BSMV03] Bart Baesens, Rudy Setiono, Christophe Mues, and Jan Vanthienen. Using neural network rule extraction and decision tables for credit-risk evaluation. *Management science*, 49(3):312–329, 2003.

[BU18] Tarek R. Besold and Sara L. Uckelman. The what, the why, and the how of artificial explanations in automated decision-making. *CoRR*, abs/1808.07074:1–20, 2018.

[BZ98] Silverio Bolognani and Mauro Zigliotto. Hardware and software effective configurations for multi-input fuzzy logic controllers. *IEEE Transactions on Fuzzy Systems*, 6(1):173–179, 1998.

[Car84] Mats Carlsson. On implementing prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[Cas21] Matteo Castigliò. Integrazione tra programmazione logica e reti neurali: esperimenti in 2P-KT. Master's thesis, Second Cycle Degree in Computer Engineering, ALMA MATER STUDIORUM—Univerisità di Bologna, 2021.

[CB15] Olivier Curé and Guillaume Blin. Chapter eight – reasoning. In *RDF Database Systems*, page 191–222. Morgan Kaufmann, Boston, 2015.

[CCDO19] Roberta Calegari, Giovanni Ciatto, Jason Dellaluce, and Andrea Omicini. Interpretable narrative explanation for ML predictors with LP: A case study for XAI. In Federico Bergenti and Stefania Monica, editors, *WOA 2019 – 20th Workshop "From Objects to Agents"*, volume 2404 of *CEUR Workshop Proceedings*, pages 105–112, Parma, Italy, 26–28 June 2019. Sun SITE Central Europe, RWTH Aachen University.

[CCDO20] Roberta Calegari, Giovanni Ciatto, Enrico Denti, and Andrea Omicini. Logic-based technologies for intelligent systems: State of the art and perspectives. *Information*, 11(3):1–29, March 2020. Special Issue "10th Anniversary of Information—Emerging Research Challenges".

[CCM$^+$18] Roberta Calegari, Giovanni Ciatto, Stefano Mariani, Enrico Denti, and Andrea Omicini. LPaaS as micro-intelligence: Enhancing IoT with symbolic reasoning. *Big Data and Cognitive Computing*, 2(3), 2018.

[CCMO21a] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini. Logic-based technologies for multi-agent systems: A systematic literature review. *Autonomous Agents and Multi-Agent Systems*, 35(1):1:1–1:67, 2021. Collection "Current Trends in Research on Software Agents and Agent-Based Software Development".

[CCMO21b] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini. Logic-based technologies for multi-agent systems: Summary of a systematic literature review. In *20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2021)*, pages 1721–1723, May 2021.

[CCN+21] Davide Calvaresi, Giovanni Ciatto, Amro Najjar, Reyhan Aydoğan, Leon Van der Torre, Andrea Omicini, and Michael Schumacher. EXPECTATION: Personalized explainable artificial intelligence for decentralized agents with heterogeneous knowledge. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Explainable and Transparent AI and Multi-Agent Systems. Third International Workshop, EXTRAAMAS 2021, Virtual Event, May 3–7, 2021, Revised Selected Papers*, volume 12688 of *Lecture Notes in Computer Science*, pages 331–343. Springer Nature, Basel, Switzerland, 2021.

[CCO20] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. *Intelligenza Artificiale*, 14(1):7–32, 2020.

[CCO21a] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. 2P-Kt: A logic-based ecosystem for symbolic AI. *SoftwareX*, 16:100817:1–7, December 2021.

[CCO21b] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. Lazy stream manipulation in Prolog via backtracking: The case of 2P-KT. In Wolfgang Faber, Gerhard Friedrich, Martin Gebser, and Michael Morak, editors, *Logics in Artificial Intelligence*, volume 12678 of *Lecture Notes in Computer Science*, pages 407–420. Springer, 2021. 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings.

[CCOC19] Giovanni Ciatto, Roberta Calegari, Andrea Omicini, and Davide Calvaresi. Towards XMAS: eXplainability through Multi-Agent Systems. In Claudio Savaglio, Giancarlo Fortino, Giovanni Ciatto, and Andrea Omicini, editors, *AI&IoT 2019 – Artificial Intelligence and Internet of Things 2019*, volume 2502 of *CEUR Workshop Proceedings*, pages 40–53. Sun SITE Central Europe, RWTH Aachen University, November 2019.

[CCS+16] Oscar Castillo, Leticia Cervantes, Jose Soria, Mauricio Sanchez, and Juan R. Castro. A generalized type-2 fuzzy granular approach with applications to aerospace. *Information Sciences*, 354:165–177, 2016.

[CCS+20] Giovanni Ciatto, Roberta Calegari, Enrico Siboni, Enrico Denti, and Andrea Omicini. 2P-KT: logic programming with objects & functions in kotlin. In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *WOA 2020 – 21th Workshop "From Objects to Agents"*, volume 2706 of *CEUR Workshop Proceedings*, pages 219–236, Aachen, Germany, October 2020. Sun SITE Central Europe, RWTH Aachen University. 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 September 2020. Proceedings.

[CCSO20] Giovanni Ciatto, Davide Calvaresi, Michael I. Schumacher, and Andrea Omicini. An abstract framework for agent-based explanations in AI. In *19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1816–1818, Auckland, New Zeland, May 2020. International Foundation for Autonomous Agents and Multiagent Systems. Extended Abstract.

[CD20] Andrew Cropper and Sebastijan Dumancic. Inductive logic programming at 30: a new introduction. *CoRR*, abs/2008.07912, 2020.

[CDDO18] Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Extending logic programming with labelled variables: Model and semantics. *Fundamenta Informaticae*, 161(1-2):53–74, July 2018. Special Issue CILC 2016.

[CDGF+95] Alessandra Costa, Alessandro De Gloria, Paolo Faraboschi, Andrea Pagni, and G. I. A. N. G. U. I. D. O. Rizzotto. Hardware solutions for fuzzy control. *Proceedings of the IEEE*, 83(3):422–434, 1995.

[CDM20] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4833–4839. ijcai.org, 2020.

[CDMO18] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Logic programming as a service. *Theory and Practice of Logic Programming*, 18(5-6):846–873, September 2018. Special Issue “Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming”.

[CDMSL$^+$20] Giovanni Ciatto, Giovanna Di Marzo Serugendo, Maxime Louvel, Stefano Mariani, Andrea Omicini, and Franco Zambonelli. Twenty years of coordination technologies: COORDINATION contribution to the state of art. *Journal of Logical and Algebraic Methods in Programming*, 113:1–25, June 2020.

[CFPP96] V. Catania, G. Ficili, S. Palazzo, and D. Panno. Using fuzzy logic in atm source traffic control: Lessons and perspectives. *IEEE Communications Magazine*, 34(11):70–74, 1996.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001.

[CH94] William W. Cohen and Haym Hirsh. Learning the classic description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning*, pages 121–133. Elsevier, 1994.

[Chu17] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

[Cia94] Paolo Ciancarini. Distributed programming with Logic Tuple Spaces. *New Generation Computing*, 12(3):251–283, 1994.

[Cia96] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[Cia21] Giovanni Ciatto. Travelling salesman problem (TSP) in 2P-KT. `https://github.com/tuProlog/ortools-tsp-example`, 2021.

[Cim06] Philipp Cimiano. *Ontology Learning and Population from Text.* Springer US, 2006.

[Cla77] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d’études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[CLW69] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. The fast fourier transform and its applications. *IEEE Transactions on Education*, 12(1):27–34, 1969.

[CM12] Mats Carlsson and Per Mildner. Sicstus prolog - the first 25 years. *Theory Pract. Log. Program.*, 12(1–2):35–66, 2012.

[CM15] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015.

[CM20] Bhavna Chilwal and P. K. Mishra. A survey of fuzzy logic inference system and other computing techniques for agricultural diseases. In Geetam Singh Tomar, Narendra S. Chaudhari, Jorge Luis V. Barbosa, and Mahesh Kumar Aghwariya, editors, *International Conference on Intelligent Computing and Smart Communication 2019*, pages 1–6, Singapore, 2020. Springer.

[CMO$^+$18] Giovanni Ciatto, Stefano Mariani, Andrea Omicini, Franco Zambonelli, and Maxime Louvel. Twenty years of coordination technologies: State-of-the-art and perspectives. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, volume 10852 of *Lecture Notes in Computer Science*, pages 51–80. Springer, 2018. 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings.

[CMOZ20] Giovanni Ciatto, Stefano Mariani, Andrea Omicini, and Franco Zambonelli. From agents to blockchain: Stairway to integration. *Applied Sciences*, 10(21):7460:1–7460:22, 2020. Special Issue “Advances in Blockchain Technology and Applications 2020”.

[CMR$^+$19] Angelo Croatti, Sara Montagna, Alessandro Ricci, Emiliano Gamberini, Vittorio Albarello, and Vanni Agnoletti. BDI personal medical assistant agents: The case of trauma tracking and alerting. *Artificial Intelligence in Medicine*, 96:187–197, 2019.

[CNVC16] Alberto Cano, Dat T. Nguyen, Sebastián Ventura, and Krzysztof J. Cios. ur-caim: improved CAIM discretization for unbalanced and balanced data. *Soft Comput.*, 20(1):173–188, 2016.

[Col86] Alain Colmerauer. Theoretical model of prolog ii. In M. van Canegham and D.-H.D. Warren, editors, *Logic Programming and its applications*, pages 3–31. Ablex Publishing Corporation, 1986.

[COS21] Roberta Calegari, Andrea Omicini, and Giovanni Sartor. Explainable and ethical AI: A perspective on argumentation and logic programming. In Matteo Baldoni and Stefania Bandini, editors, *AIxIA 2020 – Advances in Artificial Intelligence*, volume 12414 of *Lecture Notes in Computer Science*, pages 19–36. Springer Nature, 2021.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II)*, pages 37–52. ACM, April 1993.

[Cra16] Kate Crawford. Artificial intelligence's white guy problem. *The New York Times*, 25, 2016.

[CRD12] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP prolog system. *Theory Pract. Log. Program.*, 12(1-2):5–34, 2012.

[CROM19] Giovanni Ciatto, Lorenzo Rizzato, Andrea Omicini, and Stefano Mariani. TuSoW: Tuple spaces for edge computing. In *The 28th International Conference on Computer Communications and Networks (ICCCN 2019)*, Valencia, Spain, 29 July–1 August 2019. IEEE.

[CS94] Mark W. Craven and Jude W. Shavlik. Using sampling and queries to extract rules from trained neural networks. In William W. Cohen and Haym Hirsh, editors, *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, pages 37–45. Morgan Kaufmann, 1994.

[CS95] Mark W. Craven and Jude W. Shavlik. Extracting tree-structured representations of trained networks. In David S. Touretzky, Michael Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8, NIPS, Denver, CO, USA, November 27-30, 1995*, NIPS'95, pages 24–30, Cambridge, MA, USA, 1995. MIT Press.

[CSOC20] Giovanni Ciatto, Michael I. Schumacher, Andrea Omicini, and Davide Calvaresi. Agent-based explanations in ai: Towards an abstract framework. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Explainable, Transparent Autonomous Agents and Multi-Agent Systems*, volume 12175 of *Lecture Notes in Computer Science*, pages 3–20. Springer, Cham, 2020. Second International Workshop, EXTRAAMAS 2020, Auckland, New Zealand, May 9–13, 2020, Revised Selected Papers.

[CV98] Brian Center and Brahm P. Verma. Fuzzy logic for biological and agricultural systems. In *Artificial Intelligence for Biology and Agriculture*, volume 12, pages 213–225. Springer, 1998.

[CV13] Konstantina Chrysafiadi and Maria Virvou. Persiva: An empirical evaluation method of a student model of an intelligent e-learning environment for computer programming. *Computers & Education*, 68:322–333, 2013.

[CW96] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[CYLT05] Kuo-Ming Chao, Muhammad Younas, Chi-Chun Lo, and Tao-Hsin Tan. Fuzzy matchmaking for web services. In *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, volume 2, pages 721–726. IEEE, 2005.

[DCC21] Jason Dellaluce, Roberta Calegari, and Giovanni Ciatto. Probabilistic logic programming in 2p-kt. In Viviana Mascardi, Matteo Palmonari, and Giuseppe Vizzari, editors, *AIxIA 2021 Discussion Papers*, volume 3078 of *CEUR Workshop Proceedings*, pages 19–32. Sun SITE Central Europe, RWTH Aachen University, dec 2021.

[Del21] Jason Dellaluce. Enhancing symbolic AI ecosystems with probabilistic logic programming: a kotlin multi-platform case study. Master's thesis, Second Cycle Degree in Computer Engineering, Alma Mater Studiorum—Univerisità di Bologna, 2021.

[dFM15] Enric Junque de Fortuny and David Martens. Active learning-based pedagogical rule extraction. *IEEE Transactions on Neural Networks and Learning Systems*, 26(11):2664–2677, November 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[dGBG01] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artif. Intell.*, 125(1-2):155–207, 2001.

[DGS17] Michelangelo Diligenti, Marco Gori, and Claudio Saccà. Semantic-based regularization for learning and inference. *Artif. Intell.*, 244:143–165, 2017.

[dGZ99] Artur S. d'Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Appl. Intell.*, 11(1):59–77, 1999.

[dK15] Luc de Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[dKL98] Mark d'Inverno, D. Kinney, and Michael Luck. Interaction protocols in agentis. In *Proceedings International Conference on Multi Agent Systems (Cat. No. 98EX160)*, pages 112–119. IEEE, 1998.

[DKS95] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In Armand Prieditis and Stuart J. Russell, editors, *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, USA, July 9-12, 1995*, pages 194–202. Morgan Kaufmann, 1995.

[DKT09] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Assumption-based argumentation. In *Argumentation in artificial intelligence*, pages 199–218. Springer, 2009.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DMGM$^+$13] Dario Della Monica, Valentin Goranko, Angelo Montanari, Guido Sciavicco, et al. Interval temporal logics: a journey. *Bulletin of EATCS*, 3(105), 2013.

[DMRT06] Francesco M. Donini, Marina Mongiello, Michele Ruta, and Rodolfo Totaro. A model checking-based method for verifying web application design. *Electronic Notes in Theoretical Computer Science*, 151(2):19–32, 2006.

[DOC13] Enrico Denti, Andrea Omicini, and Roberta Calegari. tuProlog: Making Prolog ubiquitous. *ALP Newsletter*, October 2013.

[DOR01] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuprolog. *Science of Computer Programming*, 57(2):217–250, August 2005.

[DR08a] Luc De Raedt. *Logical and relational learning.* Springer Science & Business Media, 2008.

[DR$^+$08b] Xu Dong, Bondugula Rajkumar, et al. *Applications of fuzzy logic in bioinformatics*, volume 9. World Scientific, 2008.

[DRG17] Michelangelo Diligenti, Soumali Roychowdhury, and Marco Gori. Integrating prior knowledge into deep learning. In Xuewen Chen, Bo Luo, Feng Luo, Vasile Palade, and M. Arif Wani, editors, *16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, Cancun, Mexico, December 18-21, 2017*, pages 920–923. IEEE, 2017.

[DRK10] Luc De Raedt and Kristian Kersting. *Statistical Relational Learning*, pages 916–924. Springer US, Boston, MA, 2010.

[DRKT07] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12*, pages 2462–2467, 2007.

[DRW96] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 117—126, New York, NY, USA, 1996. Association for Computing Machinery.

[Dun95] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, September 1995.

[DVK17] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *CoRR*, abs/1702.08608, 2017.

[EEM21] Hanan Elhilbawi, Seif Eldawlatly, and Hani Mahdi. The importance of discretization methods in machine learning applications: A case study of predicting ICU mortality. In Aboul Ella Hassanien, Kuo-Chi Chang, and Mincong Tang, editors, *Advanced Machine Learning Technologies and Applications - Proceedings of AMLTA 2021, Cairo, Egypt, March 22-24, 2021*, volume 1339 of *Advances in Intelligent Systems and Computing*, pages 214–224. Springer, 2021.

[EFTW04] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*, volume 2923 of *Lecture Notes in Computer Science*, pages 87–99. Springer, 2004.

[EG18] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, January 2018.

[EIST05] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 90–96. Professional Book Center, 2005.

[Ell19] Anthony Elliott. *The Culture of AI: Everyday Life and the Digital Revolution*. Routledge, 2019.

[FdBR$^+$15] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[FDH01] Ray J. Frank, Neil Davey, and Stephen P. Hunt. Time series prediction and neural networks. *J. Intell. Robotic Syst.*, 31(1-3):91–103, 2001.

[FGKGP09] Georgios E. Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009.

[FH17] Marion Fourcade and Kieran Healy. Categories all the way down. *Historical Social Research/Historische Sozialforschung*, pages 286–296, 2017.

[FIP02] Foundation for Intelligent Physical Agents (FIPA). *Agent Communication Language Specifications*, 2002.

[FJKG10] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1988–1993. IEEE, 2010.

[FK97] Tze Ho Fung and Robert Kowalski. The IFF proof procedure for abductive logic programming. *The Journal of Logic Programming*, 33(2):151–165, 1997.

[FL08] Alexander Ferrein and Gerhard Lakemeyer. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991, 2008.

[FMDS14] Stan Franklin, T. Madl, S. D'Mello, and J. Snaider. Lida: A systems-level architecture for cognition, emotion, and learning. *IEEE Transactions on Autonomous Mental Development*, 6(1):19–41, March 2014.

[FML$^+$19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 417–426. ACM, 2019.

[Fre14] Alex A. Freitas. Comprehensible classification models: a position paper. *ACM SIGKDD Explorations Newsletter*, 15(1):1–10, June 2014.

[Frü98] Thom W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[FSM$^+$07] Leonardo Franco, José Luis Subirats, Ignacio Molina, Emilio Alba, and José M. Jerez. Early breast cancer prognosis prediction and rule extraction using a new constructive neural network algorithm. In *International Work-Conference on Artificial Neural Networks*, pages 1004–1011. Springer, 2007.

[Fu94] LiMin Fu. Rule generation from neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(8):1114–1124, 1994.

[FW99] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[FZdG14] Manoel V. M. França, Gerson Zaverucha, and Artur S. d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94(1):81–104, 2014.

[Gal85] Jean H. Gallier. *Logic for computer science: foundations of automatic theorem proving.* Harper & Row Publishers, Inc., USA, 1985.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning.* Adaptive computation and machine learning. MIT Press, 2016.

[GC06] Tao Gong and Zixing Cai. An immune agent for web-based ai course. *International Journal on E-Learning*, 5(4):493–506, 2006.

[GCS13] Sergio Alejandro Gómez, Carlos Iván Chesñevar, and Guillermo Ricardo Simari. Ontoarg: A decision support framework for ontology integration based on argumentation. *Expert Systems with Applications*, 40(5):1858–1870, 2013.

[GDF19] Travis R. Goodwin and Dina Demner-Fushman. Bridging the knowledge gap: Enhancing question answering with world and domain knowledge. *arXiv preprint arXiv:1910.07429*, 2019.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GF17] Bryce Goodman and Seth Flaxman. European Union regulations on algorithmic decision-making and a "right to explanation". *AI Magazine*, 38(3):50–57, 2017.

[GFHS04] Cui Guangzuo, Chen Fei, Chen Hu, and Li Shufang. Ontoedu: a case study of ontology-based education grid system for e-learning. In *GCCCE2004 International conference*, pages 1–9, Hong Kong, February 2004.

[GG12] Sanjeev Goyal and Sandeep Grover. Applying fuzzy grey relational analysis for ranking the advanced manufacturing systems. *Grey Systems: Theory and Application*, 2(2):284–298, 2012.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

[Gio21] Andrea Giordano. Extending the 2P-KT ecosystem with concurrent logic programming support. Master's thesis, Second Cycle Degree in Computer Science and Engineering, ALMA MATER STUDIORUM—Univerisità di Bologna, 2021.

[GL05] Anna Maria Gil-Lafuente. *Fuzzy logic in financial analysis*. Springer, 2005.

[GLMW18] Sara A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. Summary report of the second international competition on computational models of argumentation. *AI Magazine*, 39(4):77–79, December 2018.

[GMR^{+}19] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5):93:1–93:42, 2019.

[GMS04] Valentin Goranko, Angelo Montanari, and Guido Sciavicco. A road map of interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics*, 14(1–2):9–54, 2004.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SoftwareE - Practice and Experience*, 30(11):1203–1233, 2000.

[GO93] C. Lee Giles and Christian W. Omlin. Rule refinement with recurrent neural networks. In *IEEE International Conference on Neural Networks*, pages 801–806. IEEE, 1993.

[Goz12] Kerim Goztepe. Designing fuzzy rule based expert system for cyber security. *International Journal of Information Security Science*, 1(1):13–19, 2012.

[GPM^{+}14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2672–2680, 2014.

[GR68] C. Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *1968 23rd ACM National Conference*, pages 169–181, 1968.

[GR10] Marco Gavanelli and Francesca Rossi. Constraint logic programming. In Agostino Dovier and Enrico Pontelli, editors, *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP*, volume 6125 of *LNCS*, pages 64–86. Springer, 2010.

[GRSC98] Sumit Ghosh, Qutaiba Razouqi, H. Jerry Schumacher, and Aivars Celmins. A survey of recent advances in fuzzy logic in telecommunications networks and new challenges. *IEEE Transactions on Fuzzy Systems*, 6(3):443–447, 1998.

[GS04] Alejandro J. Garcia and Guillermo R. Simari. Defeasible logic programming: An argumentative approach. *Theory and practice of logic programming*, 4(1–2):95–138, 2004.

[GSS15] Yogesh Gupta, Ashish Saini, and A. K. Saxena. A new fuzzy logic based ranking function for efficient information retrieval system. *Expert Systems with Applications*, 42(3):1223–1234, 2015.

[Gun16] David Gunning. Explainable artificial intelligence (XAI). Funding Program DARPA-BAA-16-53, DARPA, 2016.

[GWW$^+$16] Shu Guo, Quan Wang, Lihong Wang, Bin Wang, and Li Guo. Jointly embedding knowledge graphs and logical rules. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 192–202, 2016.

[H2O16] H2O.ai. *H2O*, october 2016.

[Han06] David J Hand. Data mining. *Encyclopedia of Environmetrics*, 2, 2006.

[HBV06] Johan Huysmans, Bart Baesens, and Jan Vanthienen. ITER: An algorithm for predictive regression rule extraction. In *Data Warehousing and Knowledge Discovery (DaWaK 2006)*, pages 270–279. Springer, 2006.

[HCCL05] Yih-Jen Horng, Shyi-Ming Chen, Yu-Chuan Chang, and Chia-Hoang Lee. A new method for fuzzy information retrieval based on fuzzy hierarchical clustering and fuzzy inference techniques. *IEEE Transactions on Fuzzy Systems*, 13(2):216–228, 2005.

[HDM$^+$11] Johan Huysmans, Karel Dejaeger, Christophe Mues, Jan Vanthienen, and Bart Baesens. An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models. *Decision Support Systems*, 51(1):141–154, 2011.

[Hel19] Dirk Helbing. Societal, economic, ethical and legal challenges of the digital revolution: From big data to deep learning, artificial intelligence, and manipulative technologies. In *Towards Digital Enlightenment*, pages 47–72. Springer, 2019.

[Hen01] James Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37, March 2001.

[Hen08] J. Hendler. Avoiding another ai winter. *IEEE Intelligent Systems*, 23(2):2–4, March 2008.

[HHH07] Barbara Hammer, Barbara Hammer, and Pascal Hitzler. *Perspectives of Neural-Symbolic Integration*, volume 77. Springer Publishing Company, Incorporated, 1st edition, 2007.

[HHK00] James Hollan, Edwin Hutchins, and David Kirsh. Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7:174–196, 2000.

[HLW08] Steffen Hölldobler, Carsten Lutz, and Heinrich Wansing. *Logics in Artificial Intelligence*, volume 5293. Springer, 2008.

[HML$^+$16] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2410–2420, 2016.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[Hor05] Ian Horrocks. OWL: A description logic based ontology language. In Peter Van Beek, editor, *Principles and Practice of Constraint Programming (CP 2005)*, pages 5–8. Springer, 2005. Extended Abstract.

[Hor16] Paul Horwich. *Probability and evidence.* Cambridge University Press, 2016.

[HQR17] Robert Hoehndorf and Núria Queralt-Rosinach. Data science and symbolic ai: Synergies, challenges and opportunities. *Data Science*, 2017.

[HRHL01] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Intelligent agents-summary of an agent infrastructure. In *Proceedings of the 5th International conference on autonomous agents*, 2001.

[HS97] K. M. Ho and Paul D. Scott. Zeta: A global method for discretization of continuous variables. In David Heckerman, Heikki Mannila, and Daryl Pregibon, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997*, pages 191–194. AAAI Press, 1997.

[HSS03] Alexander Hofmann, Carsten Schmitz, and Bernhard Sick. Rule extraction from neural networks for intrusion detection in computer networks. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, volume 2, pages 1259–1265. IEEE, 2003.

[HSY00] Yoichi Hayashi, Rudy Setiono, and Katsumi Yoshida. A comparison between two neural network rule extraction techniques for the diagnosis of hepatobiliary disorders. *Artificial intelligence in Medicine*, 20(3):205–216, 2000.

[Hub99] Marcus J. Huber. Jam: A bdi-theoretic mobile agent architecture. In *Proceedings of the third annual conference on Autonomous Agents*, pages 236–243, 1999.

[HZC21] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowl. Based Syst.*, 212:106622, 2021.

[ISO00] ISO/IEC JTC 1/SC 22 Technical Committee. ISO/IEC 13211-2:2000: Information technology — programming languages — Prolog — part 2: Modules. International Standard ISO/IEC 13211-2, ISO/IEC, 2000.

[JDV14] Andreas Schmidt Jensen, Virginia Dignum, and Jorgen Villadsen. The AORTA architecture: Integrating organizational reasoning in jason. In Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *Lecture Notes in Computer Science*, pages 127–145. Springer International Publishing, Cham, 2014.

[JL87] Joxan Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, October 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JS13] Dana A Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Computing*, 39(1):1–20, 2013.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu, editors, *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 675–678. ACM, 2014.

[JWHY15] Jianmin Ji, Hai Wan, Ziwei Huo, and Zhenfeng Yuan. Simplifying A logic program using its consequences. In Qiang Yang and Michael J. Wooldridge, editors, *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 3069–3075, Buenos Aires, Argentina, 25–31 July 2015. AAAI Press.

[KB07] Marius Kloetzer and Calin Belta. Temporal logic planning and control of robotic swarms by hierarchical abstractions. *IEEE Transactions on Robotics*, 23(2):320–330, March 2007.

[KBB+21] Philipp Körner, Michael Beuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. 50 years of prolog and beyond. *Theory and Practice of Logic Programming*, 2021. (Currently under review).

[KC04] Lukasz A. Kurgan and Krzysztof J. Cios. CAIM discretization algorithm. *IEEE Trans. Knowl. Data Eng.*, 16(2):145–153, 2004.

[KDDR+11] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11(2–3):235–262, 2011.

[Ker92] Randy Kerber. Chimerge: Discretization of numeric attributes. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*, pages 123–128. AAAI Press / The MIT Press, 1992.

[KFQK21] Eoin M. Kenny, Courtney Ford, Molly Quinn, and Mark T. Keane. Explaining black-box classifiers using post-hoc explanations-by-example: The effect of explanations and error-rates in xai user studies. *Artificial Intelligence*, 294:103459, 2021.

[KHC18] Tanel Kerikmae, Thomas Hoffmann, and Archil Chochia. Legal technology for law firms: determining roadmaps for innovation. *Croatian International Relations Review*, 24(81):91–112, 2018.

[KJN08] Rikard Konig, Ulf Johansson, and Lars Niklasson. G-REX: A versatile framework for evolutionary data mining. In *2008 IEEE International Conference on Data Mining Workshops (ICDM 2008 Workshops)*, pages 971–974, 2008.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, 2002.

[Kot07] Sotiris Kotsiantis. Supervised machine learning: A review of classification techniques. In *Emerging Artificial Intelligence Applications in Computer Engineering*, volume 160 of *Frontiers in Artificial Intelligence and Applications*, pages 3–24. IOS Press, October 2007.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress*, pages 569–574. North-Holland, August 1974.

[Kow79] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, jul 1979.

[KSB99] R. Krishnan, G. Sivakumar, and P. Bhattacharya. Extracting decision trees from trained neural networks. *Pattern Recognition*, 32(12):1999–2009, 1999.

[KYA$^+$16] Alexander Kohan, Mitsuharu Yamamoto, Cyrille Artho, Yoriyuki Yamagata, Lei Ma, Masami Hagiya, and Yoshinori Tanabe. Java pathfinder on android devices. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.

[KYZ00] Janusz Kacprzyk, Ronald R. Yager, and S. Zadrożny. A fuzzy logic based approach to linguistic summaries of databases. *International Journal of Applied Mathematics and Computer Science*, 10(4):813–834, 2000.

[KZZ89] Janusz Kacprzyk, Sławomir Zadrożny, and Andrzej Ziołkowski. Fquery iii+: a “human-consistent” database querying system based on fuzzy logic with linguistic quantifiers. *Information Systems*, 14(6):443–453, 1989.

[Lac10] Nicolas Lachiche. *Propositionalization*, pages 812–817. Springer US, Boston, MA, 2010.

[LB87] Hector J. Levesque and Ronald J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Comput. Intell.*, 3:78–93, 1987.

[LD94] Jaeho Lee and Edmund H. Durfee. Structured circuit semantics for reactive plan execution systems. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 2, pages 1232–1237, Seattle, WA, USA, 31 July—4 August 1994. AAAI Press / The MIT Press.

[Len95] Douglas B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[Lev84] Hector J. Levesque. A logic of implicit and explicit belief. In *4th AAAI Conference on Artificial Intelligence (AAAI '84)*, number 5 in AAAI'84, pages 198–202, Austin, Texas, 1984. AAAI Press.

[Lip18] Zachary C. Lipton. The mythos of model interpretability. *Commun. ACM*, 61(10):36–43, 2018.

[LKR$^+$16] Paulo Leitão, Stamatis Karnouskos, Luis Ribeiro, Jay Lee, Thomas I. Strasser, and Armando W. Colombo. Smart agents in industrial cyber-physical systems. *Proceedings of the IEEE*, 104(5):1086–1101, 2016.

[LL05] Jiang-Long Lin and C. L. Lin. The use of grey-fuzzy logic for the optimization of the manufacturing process. *Journal of Materials Processing Technology*, 160(1):9–14, 2005.

[LL09] Kevin F. R. Liu and Jia-Hong Lai. Decision-support for environmental impact assessment: A hybrid approach using fuzzy logic and fuzzy analytic network process. *Expert Systems with Applications*, 36(3):5119–5136, 2009.

[Llo90] John W Lloyd. *Computational logic*. Springer, 1990.

[LLS02] Hugo Liu, Henry Lieberman, and Ted Selker. Goose: a goal-oriented search engine with commonsense. In Conejo R. De Bra P., Brusilovsky P., editor, *International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, volume 2347 of *Lecture Notes in Computer Science*, pages 253–263. Springer, 2002.

[LLSB04] Henry Lieberman, Hugo Liu, Push Singh, and Barbara Barry. Beating common sense into interactive applications. *AI Magazine*, 25(4):63–63, 2004.

[LMS14] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. A thread-safe library for binary decision diagrams. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5*, volume 8702 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2014.

[LMVW11] Ondrej Linda, Milos Manic, Todd Vollmer, and Jason Wright. Fuzzy logic based anomaly detection for embedded network security cyber sensor. In *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 202–209. IEEE, 2011.

[LR07] James R Larus and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 1(1):1–226, 2007.

[LS04] Hugo Liu and Push Singh. Conceptnet–a practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226, 2004.

[Lyt08] Miltiadis D. Lytras. *Knowledge Management Strategies: A Handbook of Applied Technologies: A Handbook of Applied Technologies*, volume 5. IGI Global, 2008.

[Mak87] J. A. Makowsky. Why horn formulas matter in computer science: Initial structures and generic examples. *Journal of Computer and System Sciences*, 34(2):266–292, 1987.

[MB88] Stephen Muggleton and Wray L. Buntine. Machine invention of first order predicates by inverting resolution. In John E. Laird, editor, *Machine Learning, Proceedings of the Fifth International*

Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988, pages 339–352. Morgan Kaufmann, 1988.

[MBVGV07] David Martens, Bart Baesens, Tony Van Gestel, and Jan Vanthienen. Comprehensible credit scoring models using rule extraction from support vector machines. *European Journal of Operational Research*, 183(3):1466–1476, 2007.

[MC94] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994.

[McC89] John McCarthy. Artificial intelligence, logic and formalizing common sense. *Philosophical logic and artificial intelligence*, pages 161–190, 1989.

[McN77] George F. McNulty. Fragments of first order logic, i: Universal Horn logic. *Journal of Symbolic Logic*, 42(2):221–237, 1977.

[MCO22] Matteo Magnini, Giovanni Ciatto, and Andrea Omicini. Injecting first order logic formulæ into neural networks: Experiments report. In *31st International Joint Conference on Artificial Intelligence (IJCAI 2022)*, Vienna, Austria, 23–29 July 2022. AAAI Press.

[Md94] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994. Special Issue: Ten Years of Logic Programming.

[MDK$^+$18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3749–3759. Curran Associates, Inc., 2018.

[MF92] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In Stephen Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[MGDG19] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. LYRICS: a general interface layer to integrate AI and deep learning. *arXiv preprint arXiv:1903.07534*, abs/1903.07534, 2019.

[MH03] Ralf Moller and Volker Haarslev. *Description logic systems*, pages 282–305. Cambridge University Press, 2003.

[Mil56] George Abram Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, March 1956.

[Min75] Marvin Minsky. A framework for representing knowledge representation. In *The Psychology of Computer Vision.* Mc Graw-Hill, New-York (NY, US), 1975.

[Min91] Marvin Minsky. *Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy*, page 218–243. MIT Press, Cambridge, MA, USA, 1991.

[Mit97] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MLPT14] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Mach. Learn.*, 94(1):25–49, 2014.

[MLT15] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.*, 100(1):49–73, 2015.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[MMN18] Fabio Martinelli, Francesco Mercaldo, and Vittoria Nardone. Identifying insecure features in android applications using model checking. In Paolo Mori, Steven Furnell, and Olivier Camp, editors, *4th International Conference on Information Systems Security and Privacy (ICISSP 2018), Funchal, Madeira - Portugal, January 22-24, 2018*, pages 589–596. SciTePress, 2018.

[MMS92] Clayton McMillan, Michael C. Mozer, and Paul Smolensky. Rule induction through integrated symbolic and subsymbolic processing. In *Advances in neural information processing systems*, pages 969–976, 1992.

[MN96] George Metakides and Anil Nerode. *Principles of Logic and Logic Programming*, volume 13 of *Studies in Computer Science and Artificial Intelligence.* Elsevier, USA, 1996.

[MOC17] Stefano Mariani, Andrea Omicini, and Giovanni Ciatto. Novel opportunities for tuple-based coordination: XPath, the Blockchain, and stream processing. In Pasquale De Meo, Maria Nadia Postorino, Domenico Rosaci, and Giuseppe M.L. Sarné, editors, *WOA 2017 – 18th Workshop "From Objects to Agents"*, volume 1867 of *CEUR Workshop Proceedings*, pages 61–64. Sun SITE Central Europe, RWTH Aachen University, June 2017.

[MOCO06] Conor Muldoon, Gregory M. P. O'Hare, Rem Collier, and Michael J. O'Grady. Agent factory micro edition: A framework for ambient applications. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3993 of *Lecture Notes in Computer Science*, pages 727–734. Springer Berlin Heidelberg, 2006.

[MP88] Marvin L. Minsky and Seymour A. Papert. *Perceptrons: Expanded Edition.* MIT Press, Cambridge, MA, USA, 1988.

[MP91] Patrick M. Murphy and Michael J. Pazzani. Id2-of-3: Constructive induction of m-of-n concepts for discriminators in decision trees. In *Machine Learning Proceedings 1991*, pages 183–187. Elsevier, 1991.

[MP14] Sanjay Modgil and Henry Prakken. The aspic+ framework for structured argumentation: a tutorial. *Argument & Computation*, 5(1):31–62, 2014.

[MS58] John McCarthy and Claude Shannon. Automata studies. *Journal of Symbolic Logic*, 23(1):59–60, 1958.

[MTC$^+$10] Marco Montali, Paolo Torroni, Federico Chesani, Paola Mello, Marco Alberti, and Evelina Lamma. Abductive logic programming as an effective technology for the static verification of declarative business processes. *Fundamenta Informaticae*, 102(3–4):325–361, 2010.

[MTCB17] John Mbuli, Damien Trentesaux, Joffrey Clarhaut, and Guillaume Branger. Decision support in condition-based maintenance of a fleet of cyber-physical systems: a fuzzy logic approach. In *2017 Intelligent Systems Conference (IntelliSys)*, pages 82–89. IEEE, 2017.

[Mug91] Stephen Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New Gener. Comput.*, 13(3&4):245–286, 1995.

[NAC08] Haydemar Núñez, Cecilio Angulo, and Andreu Català. Rule extraction based on support and prototype vectors. In Joachim Diederich, editor, *Rule Extraction from Support Vector Machines*, volume 80 of *Studies in Computational Intelligence*, pages 109–134. Springer, 2008.

[Nil01] Nils J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5(B):99–110, 2001.

[NOV11] Elena Nardini, Andrea Omicini, and Mirko Viroli. Description spaces with fuzziness. In Mathew J. Palakal, Chih-Cheng Hung, William Chu, and W. Eric Wong, editors, *26th Annual ACM Symposium on Applied Computing (SAC 2011)*, volume II: Artificial Intelligence & Agents, Information Systems, and Software Development, pages 869–876, Tunghai University, TaiChung, Taiwan, 21–25 March 2011. ACM.

[NS92] Raymond Ng and Venkatramanan Siva Subrahmanian. Probabilistic logic programming. *Information and computation*, 101(2):150–201, 1992.

[NVP10] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with tucson semantic tuple centres. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 2037–2044, Sierre, Switzerland, March 22-26, 2010, February 2010. ACM.

[NZRS12] Feng Niu, Ce Zhang, Christopher Ré, and Jude Shavlik. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 8(3):42–73, 2012.

[OC19] Andrea Omicini and Roberta Calegari. Injecting (micro)intelligence in the IoT: Logic-based approaches for (M)MAS.

In Donghui Lin, Toru Ishida, Franco Zambonelli, and Itsuki Noda, editors, *Massively Multi-Agent Systems II*, volume 11422 of *Lecture Notes in Computer Science*, chapter 2, pages 21–35. Springer, May 2019. International Workshop, MMAS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers.

[OD01] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[ODN95] Andrea Omicini, Enrico Denti, and Antonio Natali. Agent coordination and control through logic theories. In Marco Gori and Giovanni Soda, editors, *Topics in Artificial Intelligence*, volume 992 of *LNAI*, pages 439–450. Springer-Verlag, 1995. 4th Congress of the Italian Association for Artificial Intelligence (AI*IA'95), Florence, Italy, 11–13 October 1995, Proceedings.

[OFM$^+$21] Alfonso Ortega, Julian Fierrez, Aythami Morales, Zilong Wang, and Tony Ribeiro. Symbolic ai for xai: Evaluating lfit inductive programming for fair and explainable automatic recruitment. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 78–87. IEEE, 2021.

[Omi99] Andrea Omicini. On the semantics of tuple-based coordination models. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 175–182, New York, NY, USA, 28 February – 2 March 1999. ACM.

[Omi01] Andrea Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In Paolo Ciancarini and Michael J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 185–193. Springer-Verlag, 2001. 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 June 2000. Revised Papers.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. Association for Computing Machinery.

[OP11] Mohammad Oliya and Hung Keng Pung. Towards incremental reasoning for context aware systems. In *Advances in Computing* *and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 232–241. Springer, 2011.

[Oss12] Sascha Ossowski. *Agreement technologies*, volume 8 of *Law, Governance and Technology Series*. Springer Netherlands, 2012.

[OSS15] Ming Erh Ooi, Mohd Sayuti, and Ahmed A. D. Sarhan. Fuzzy logic-based approach to investigate the novel uses of nano suspended lubrication in precise machining of aerospace al tempered grade 6061. *Journal of Cleaner Production*, 89:286–295, 2015.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

[Pau18] Lawrence C. Paulson. Computational logic: its origins and applications. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2210):20170872, 2018.

[PBOR08] Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented Prolog engine. In Roger L. Wainwright, Hisham M. Haddad, Ronaldo Menezes, and Mirko Viroli, editors, *23rd ACM Symposium on Applied Computing (SAC 2008)*, volume 1, pages 191–197, Fortaleza, Ceará, Brazil, 16–20 March 2008. ACM. Special Track on Programming Languages.

[PCOS20] Giuseppe Pisano, Roberta Calegari, Andrea Omicini, and Giovanni Sartor. Arg-`tu`Prolog: A `tu`Prolog-based argumentation framework. In Francesco Calimeri, Simona Perri, and Ester Zumpano, editors, *CILC 2020 – Italian Conference on Computational Logic. Proceedings of the 35th Italian Conference on Computational Logic*, volume 2719 of *CEUR Workshop Proceedings*, pages 51–66, Aachen, Germany, 13-15 October 2020. Sun SITE Central Europe, RWTH Aachen University, CEUR-WS.

[PF19] Laurent Perron and Vincent Furnon. OR-tools. `https://developers.google.com/optimization/`, 2019.

[PKD+12] Prakashgoud Patil, Umakant Kulkarni, B. L. Desai, V. I. Benagi, and V. B. Naragund. Fuzzy logic based irrigation control system using wireless sensor network for precision agriculture. *Agro-Informatics and Precision Agriculture (AIPA)*, 2012.

[Pla21] 2P-Kt Playground. Web interface. `https://pika-lab.gitlab.io/tuprolog/2p-kt-web`, 2021. Last access: April 17, 2022.

[Plo71] G. D. Plotkin. A further note on inductive generalization. In *Machine Intelligence 6*, pages 101–124. American Elsevier, 1971.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[Pol87] John L. Pollock. Defeasible reasoning. *Cognitive science*, 11(4):481–518, 1987.

[Pra13] Henry Prakken. *Logical tools for modelling legal argument: a study of defeasible reasoning in law*, volume 32. Springer Science & Business Media, 2013.

[Pro21a] Ciao! Prolog. Home page. `https://ciao-lang.org`, 2021. Last access: April 17, 2022.

[Pro21b] ECLiPSe Prolog. Home page. `https://eclipseclp.org`, 2021. Last access: April 17, 2022.

[Pro21c] SWI Prolog. Home page. `https://www.swi-prolog.org`, 2021. Last access: April 17, 2022.

[Pro21d] Tau Prolog. Home page. `http://tau-prolog.org`, 2021. Last access: April 17, 2022.

[Pro21e] XSB Prolog. Home page. `http://xsb.sourceforge.net`, 2021. Last access: April 17, 2022.

[PS15] Henry Prakken and Giovanni Sartor. Law and logic: A review from an argumentation perspective. *Artificial Intelligence*, 227:214–245, 2015.

[PSRV19] Francesc Pedro, Miguel Subosa, Axel Rivas, and Paula Valverde. Artificial intelligence in education: challenges and opportunities for sustainable development. Technical report, Paris, 2019.

[PVB$^+$13] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[PVG$^+$11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011.

[PW78] David Premack and Guy Woodruff. Does the chimpanzee have a theory of mind? *Behavioral and brain sciences*, 1(4):515–526, December 1978.

[QNO06] Mohammed A. Quddus, Robert B. Noland, and Washington Y. Ochieng. A high accuracy fuzzy logic based map matching algorithm for road transport. *Journal of Intelligent Transportation Systems*, 10(3):103–115, 2006.

[Qui87] J. Ross Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987.

[Qui93] J. Ross Quinlan. C4.5: Programming for machine learning. *Morgan Kauffmann*, 1993.

[Rao96] Anand S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, Berlin, Heidelberg, 1996.

[RAS15] Aakanksha Rastogi, Ritika Arora, and Shanu Sharma. Leaf disease detection and grading using computer vision technology & fuzzy logic. In *Proccedings of the 2nd international conference on signal processing and integrated networks (SPIN)*, pages 500–505. IEEE, 2015.

[Red16] Christoph Redl. The DLVHEX system for knowledge representation: recent advances (system description). *Theory and Practice of Logic Programming*, 16(5-6):866–883, 2016.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1–2):81–132, 1980.

[RHW86] D. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Rig07] Fabrizio Riguzzi. A top down interpreter for LPAD and cp-logic. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence, Rome, Italy, September 10-13, 2007, Proceedings*, volume 4733 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2007.

[Rig18] Fabrizio Riguzzi. *Foundations of Probabilistic Logic Programming.* River Publishers, Gistrup, Denmark, 2018.

[Riz18] Lorenzo Rizzato. Coordination as a web service: una moderna implementazione del modello Linda, 2018. First Cycle Degree in Computer Science and Engineering, ALMA MATER STUDIORUM—Univerisità di Bologna.

[RN16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Ros57] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para.* Cornell Aeronautical Laboratory, 1957.

[Ros00] Francesca Rossi. Constraint (logic) programming: A survey on research and applications. In Krzysztof R. Apt, Eric Monfroy, Antonis C. Kakas, and Francesca Rossi, editors, *New Trends in Constraints*, pages 40–74. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[RPM12] Anderson Rocha, Joao Paulo Papa, and Luis A. A. Meira. How far do we get using machine learning black-boxes? *International Journal of Pattern Recognition and Artificial Intelligence*, 26(02):1261001–(1–23), 2012.

[RR17] Tim Rocktaschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pages 3788–3800, 2017.

[RR19] Avi Rosenfeld and Ariella Richardson. Explainability in human-agent systems. *Autonomous Agents and Multi-Agent Systems*, 33(6):673–705, November 2019.

[RS11] Fabrizio Riguzzi and Terrance Swift. The PITA system for logical-probabilistic inference. In Stephen H. Muggleton and Hiroaki Watanabe, editors, *Latest Advances in Inductive Logic Programming, ILP 2011, Late Breaking Papers, Windsor Great Park, UK, July 31 - August 3*, pages 79–86. Imperial College Press / World Scientific, 2011.

[RSG16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ”why should I trust you?”: Explaining the predictions of any classifier. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144. ACM, 2016.

[RSR15] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1119–1129, 2015.

[Rud19] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[RVBW08] Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

[SA11] Semih Sezer and Ali Erdem Atalay. Dynamic modeling and fuzzy logic control of vibrations of a railway vehicle for different track irregularities. *Simulation Modelling Practice and Theory*, 19(9):1873–1894, 2011.

[SAG99] Gregor P. J. Schmitz, Chris Aldrich, and François S. Gouws. ANN-DT: an algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*, 10(6):1392–1401, 1999.

[Sat95] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16*, pages 715–729. MIT Press, 1995.

[SAZ+18] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezný, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.

[SBM11] Rudy Setiono, Bart Baesens, and Christophe Mues. Rule extraction from minimal neural networks for credit card screening. *International Journal of Neural Systems*, 21(04):265–276, 2011.

[SCCO21] Federico Sabbatini, Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. On the design of PSyKE: A platform for symbolic knowledge extraction. In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *WOA 2021 – 22nd Workshop "From Objects to Agents"*, volume 2963 of *CEUR Workshop Proceedings*, pages 29–48, Bologna, Italy, October 2021. Sun SITE Central Europe, RWTH Aachen University. 22nd Workshop "From Objects to Agents" (WOA 2021), Bologna, Italy, 1–3 September 2021. Proceedings.

[SCK00] Helmut Simonis, Philippe Charlier, and Philip Kay. Constraint handling in an integrated transportation problem. *IEEE Intelligent Systems and their applications*, 15(1):26–32, 2000.

[SCMN13] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934, 2013.

[SCO21] Federico Sabbatini, Giovanni Ciatto, and Andrea Omicini. GridEx: An algorithm for knowledge extraction from black-box regressors. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Explainable and Transparent AI and Multi-Agent Systems. Third International Workshop, EXTRAAMAS 2021, Virtual Event, May 3–7, 2021, Revised Selected Papers*, volume 12688 of *Lecture Notes in Computer Science*, pages 18–38. Springer Nature, Basel, Switzerland, 2021.

[SdG16] Luciano Serafini and Artur S. d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. In Tarek R. Besold, Luís C. Lamb, Luciano Serafini, and Whitney Tabor, editors, *Proceedings of the 11th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy'16) co-located with the Joint Multi-Conference on Human-Level Artificial Intelligence (HLAI 2016), New York City, NY, USA, July 16-17, 2016*, volume 1768 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[SDG17] Luciano Serafini, Ivan Donadello, and Artur d'Avila Garcez. Learning and reasoning in logic tensor networks: Theory and application to semantic image interpretation. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 125–130, New York, NY, USA, 2017. ACM.

[Sea80] John R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3(3):417–424, 1980.

[Set97] Rudy Setiono. Extracting rules from neural networks by pruning and hidden-unit splitting. *Neural Computation*, 9(1):205–225, 1997.

[SFP$^+$07] Vitaly Schetinin, Jonathan E. Fieldsend, Derek Partridge, Timothy J. Coats, Wojtek J. Krzanowski, Richard M. Everson, Trevor C. Bailey, and Adolfo Hernandez. Confident interpretation of bayesian decision tree ensembles for clinical applications. *IEEE Transactions on Information Technology in Biomedicine*, 11(3):312–319, 2007.

[SG16] Luciano Serafini and Artur S. d'Avila Garcez. Learning and reasoning with logic tensor networks. In *Conference of the Italian Association for Artificial Intelligence*, pages 334–348. Springer, 2016.

[Sha00] Stuart C. Shapiro. *SNePS: A logic for natural language understanding and commonsense reasoning*, pages 175—195. MIT Press, Cambridge, MA, USA, 2000.

[Sib19] Enrico Siboni. 2P-KT: A kotlin-based, multi-platform framework for symbolic AI. Master's thesis, Second Cycle Degree in Computer Science and Engineering, ALMA MATER STUDIORUM—Univerisità di Bologna, 2019.

[Sim96] Helmut Simonis. Application development with the chip system. In Wallace M. Kuper G., editor, *Constraint Databases and Applications*, volume 1034 of *Lecture Notes in Computer Science*, pages 1–21. Springer, Berlin, Heidelberg, 1996.

[Sim01] Helmut Simonis. Building industrial applications with constraint programming. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Hubert Comon, Claude Marché, and Ralf Treinen, editors, *Constraints in Computational Logics*, pages 271–309. Springer, 2001. Theory and Applications International Summer School (CCL '99). Gif-sur-Yvette, France, September 5–8, 1999. Revised Lecture.

[SIS15] L. Suganthi, S. Iniyan, and Anand A. Samuel. Applications of fuzzy logic in renewable energy systems–a review. *Renewable and sustainable energy reviews*, 48:585–607, 2015.

[SK97] Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 2 Volumes*, pages 1330–1339. Morgan Kaufmann, 1997.

[SK20] Zeren Shui and George Karypis. Heterogeneous molecular graph neural networks for predicting molecule properties. In Claudia Plant, Haixun Wang, Alfredo Cuzzocrea, Carlo Zaniolo, and Xindong Wu, editors, *20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020*, pages 492–500. IEEE, 2020.

[SKT04] Dimitris Skarlatos, Kleomenis Karakasis, and Athanassios Trochidis. Railway wheel fault diagnosis using a fuzzy-logic method. *Applied Acoustics*, 65(10):951–966, 2004.

[SLZ02] Rudy Setiono, Wee Kheng Leow, and Jacek M. Zurada. Extraction of rules from artificial neural networks for nonlinear regression. *IEEE Transactions on Neural Networks*, 13(3):564–577, 2002.

[SMFM05] Pilar Sancho, Iván Martínez, and Baltasar Fernández-Manjón. Semantic web technologies applied to e-learning personalization in¡ e-aula¿. *Journal of Universal Computer Science*, 11(9):1470–1481, September 2005.

[Smi21] Smile. Statistical machine intelligence and learning engine. `https://haifengl.github.io`, 2021. [Online; last accessed 11 Oct 2021].

[Smo87] P. Smolensky. Connectionist ai, symbolic ai, and the brain. *Artificial Intelligence Review*, 1(2):95–109, Jun 1987.

[Smo90] Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46(1):159–216, 1990.

[Smu68] Raymond M. Smullyan. *First-Order Logic*. New York [Etc.]Springer-Verlag, 1968.

[SN88] Kazumi Saito and Ryohei Nakano. Medical diagnostic expert system based on PDP model. In *IEEE 1988 International Conference on Neural Networks (ICNN 1988)*, volume 1, pages 255–262, 1988.

[SN02] Kazumi Saito and Ryohei Nakano. Extracting regression rules from neural networks. *Neural Networks*, 15(10):1279–1288, 2002.

[SNB$^+$08] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[SNS$^+$06] Maria Teresinha Arns Steiner, Pedro José Steiner Neto, Nei Yoshihiro Soma, Tamio Shimizu, and J. C. Nievola. Using neural network rule extraction for credit-risk evaluation. *International Journal of Computer Science and Network Security*, 6(5):6–16, 2006.

[Sow91] John F Sowa, editor. *Principles of semantic networks: Explorations in the representation of knowledge*. Morgan Kaufmann Series in Representation and Reasoning. Morgan Kaufmann Pub, May 1991.

[SP21] SICStus Prolog. Home page. `https://sicstus.sics.se`, 2021. Last access: April 17, 2022.

[Spe21] Giovanni Maria Speciale. Il ragionamento logico come forma di apprendimento: Sviluppo di un framework per ILP. Master's thesis, Second Cycle Degree in Computer Science and Engineering, Alma Mater Studiorum—Univerisità di Bologna, 2021.

[SS04] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, August 2004.

[SSS12] Amrita Sarkar, G. Sahoo, and U. C. Sahoo. Application of fuzzy logic in transport planning. *International Journal on Soft Computing*, 3(2):1, 2012.

[STCG$^+$21] Zoran Sevarac, Jon Tait, Laura Carter-Greaves, Aidan Morgan, and Valentin Steinhauer. Neuroph. `http://neuroph.sourceforge.net/index.html`, 2021.

[Sun01] R. Sun. Artificial intelligence: Connectionist and symbolic approaches. In Neil J. Smelser and Paul B. Baltes, editors, *International Encyclopedia of the Social & Behavioral Sciences*, page 783–789. Pergamon, Oxford, 2001.

[Sun05] Ron Sun, editor. *The CLARION Cognitive Architecture: Extending Cognitive Modeling to Social Simulation*, pages 79–100. Cambridge University Press, 2005.

[SVJNM16] Maria Claudia Solarte-Vasquez, Natalia Järv, and Katrin Nyman-Metcalf. Usability factors in transactional design and smart contracting. In Tanel Kerikmäe and Addi Rull, editors, *The Future of Law and eTechnologies*, pages 149–176. Springer International Publishing, Cham, 2016.

[SW12] Terrance Swift and David Scott Warren. XSB: Extending prolog with tabled logic programming. *Theory Practice of Logic Programming*, 12(1-2):157–187, 2012.

[SY96] Leon Sterling and Ümit Yalçinalp. Logic programming and software engineering—implications for software design. *The Knowledge Engineering Review*, 11(4):333–345, 1996.

[TDD78] André Thayse, Marc Davio, and Jean-Pierre Deschamps. Optimization of multivalued decision algorithms. In *Proceedings of the eighth international symposium on Multiple-valued logic, MVL*

1978, Rosemont, Illinois, USA, 1978, pages 171–178. IEEE Computer Society Press, 1978.

[Tea] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0. `https://deeplearning4j.konduit.ai/`, year=2021.

[TH12] Pejman Tahmasebi and Ardeshir Hezarkhani. A hybrid neural networks-fuzzy logic-genetic algorithm for grade estimation. *Computers & Geosciences*, 42:18–27, 2012.

[THA92] Volker Tresp, Jürgen Hollatz, and Subutai Ahmad. Network structuring and training using rule-based knowledge. In Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*, pages 871–878. Morgan Kaufmann, 1992.

[Thr95] Sebastian Thrun. Extracting rules from artificial neural networks with distributed representations. In *Advances in neural information processing systems*, pages 505–512, 1995.

[TL18] Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning. *CoRR*, abs/1806.02847, 2018.

[TN06] Angela Torres and Juan J. Nieto. Fuzzy logic in medicine and bioinformatics. *BioMed Research International*, 2006, 2006.

[TS92] Geoffrey Towell and Jude W. Shavlik. Interpretation of artificial neural networks: Mapping knowledge-based neural networks into rules. In *Advances in neural information processing systems*, pages 977–984, 1992.

[TS93] Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101, 1993.

[TSN90] Geoffrey G. Towell, Jude W. Shavlik, and Michiel O. Noordeweir. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861–866, 1990.

[TT08] Vuong Xuan Tran and Hidekazu Tsuji. Qos based ranking for web services: Fuzzy approaches. In *2008 4th International Conference on Next Generation Web Services Practices*, pages 77–82. IEEE, 2008.

[Tuo18] Ilkka Tuomi. The impact of artificial intelligence on learning, teaching, and education. Technical report, November 2018. JRC Working Papers.

[tuP21] tuProlog. Home page. `http://tuprolog.unibo.it`, 2021. Last access: April 17, 2022.

[Tur50] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(October):433–60, 1950.

[Twa10] Bhekisipho Twala. Multiple classifier application to credit risk assessment. *Expert Systems with Applications*, 37(4):3326–3336, 2010.

[TWS19] Paul Tarau, Jan Wielemaker, and Tom Schrijvers. Lazy stream programming in Prolog. *Electronic Proceedings in Theoretical Computer Science*, 306:224–237, September 2019.

[Val05] Andre Valente. Types and roles of legal ontologies. In V. Richard Benjamins, Pompeu Casanovas, Joost Breuker, and Aldo Gangemi, editors, *Law and the semantic web*, pages 65–76. Springer, 2005.

[VBD01] Johan Van Benthem and Kees Doets. *Higher-Order Logic*, pages 189–243. Springer Netherlands, Dordrecht, 2001.

[VDB09] Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009.

[VEBB+08] Tom Van Engers, Alexander Boer, Joost Breuker, André Valente, and Radboud Winkels. Ontologies in the legal domain. In *Digital Government*, pages 233–261. Springer, 2008.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[vG90] Tim van Gelder. Why distributed representation is inherently non-symbolic. In Georg Dorffner, editor, *Konnektionismus in Artificial Intelligence und Kognitionsforschung. Proceedings 6. Österreichische Artificial Intelligence-Tagung (KONNAI), Salzburg, Österreich, 18. bis 21. September 1990*, volume 252 of *Informatik-Fachberichte*, pages 58–66. Springer, 1990.

[Vit06] Andrew J. Viterbi. A personal history of the viterbi algorithm. *IEEE Signal Process. Mag.*, 23(4):120–142, 2006.

[VO06] Mirko Viroli and Andrea Omicini. Coordination as a service. *Fundamenta Informaticae*, 73(4):507–534, 2006.

[vRMG$^+$19] Laura von Rüden, Sebastian Mayer, Jochen Garcke, Christian Bauckhage, and Jannis Schücker. Informed machine learning - towards a taxonomy of explicit integration of knowledge into machine learning. *CoRR*, abs/1903.12394, 2019.

[VRVdBDR14] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *Proceedings Workshop on Probabilistic Logic Programming (PLP)*, pages 1–10, 2014.

[VVB04] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In James P. Delgrande and Torsten Schaub, editors, *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings*, pages 409–415, 2004.

[VvdB17] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR). A Practical Guide.* Springer, 2017.

[VVL19] F. Van Veen and S. Leijnen. The neural network zoo. `https://www.asimovinstitute.org/neural-network-zoo`, 2019. [Online; accessed 17-September-2021].

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1–2):139–168, 1996.

[War83] David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition.* Morgan Kaufmann, Elsevier, 2011.

[Wik21a] Wikipedia contributors. Activation function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Activation_function`, 2021. [Online; accessed 23-August-2021].

[Wik21b] Wikipedia contributors. Decision tree learning — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Decision_tree_learning`, 2021. [Online; accessed 17-September-2021].

[Wik21c] Wikipedia contributors. Stochastic gradient descent — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Stochastic_gradient_descent`, 2021. [Online; accessed 23-August-2021].

[Win] Michael Winikoff. Jack™ intelligent agents: An industrial strength platform. chapter 7, pages 175–193.

[Win05] Michael Winikoff. Jack^{tm} intelligent agents: An industrial strength platform. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer, 2005.

[WL20] Hongwei Wang and Jure Leskovec. Unifying graph convolutional neural networks and label propagation. *CoRR*, abs/2002.06755, 2020.

[WM97] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.*, 1(1):67–82, 1997.

[WMWG17] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Trans. Knowl. Data Eng.*, 29(12):2724–2743, 2017.

[WPC$^+$21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks Learning Systems*, 32(1):4–24, 2021.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory Pract. Log. Program.*, 12(1-2):67–96, 2012.

[Wu17] Hao Wu. *Industrial Applications of Probabilistic Model Checking-A Model-based Approach for Embedded Networked Systems and Concurrent Data Structures* -. PhD thesis, RWTH Aachen University, Germany, 2017.

[WWG15] Quan Wang, Bin Wang, and Li Guo. Knowledge base completion using embeddings and rules. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[WZL$^+$15] Zhuoyu Wei, Jun Zhao, Kang Liu, Zhenyu Qi, Zhengya Sun, and Guanhua Tian. Large-scale knowledge base completion: Inferring via grounding network sampling over selected instances. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1331–1340. ACM, 2015.

[XZF$^+$18] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A semantic loss function for deep learning with symbolic knowledge. In Jennifer Dy and Andreas Krause, editors, *35th Conference on Machine Learning (ICML 2018)*, volume 80 of *Proceedings of Machine Learning Research*, pages 5502–5511. PMLR, 2018.

[Yac05] Kalina Yacef. The logic-ita in the classroom: a medium scale experiment. *International Journal of Artificial Intelligence in Education*, 15(1):41–62, 2005.

[YH12] Bingchuan Yuan and John Herbert. Fuzzy cara - a fuzzy-based context reasoning system for pervasive healthcare. *Procedia Computer Science*, 10:357–365, 2012.

[YKZ03] Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In Schmidt D.C. Meersman R., Tari Z., editor, *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, volume 2888 of *Lecture Notes in Computer Science*, pages 671–688. Springer Berlin Heidelberg, 2003.

[YL99] John Yen and Reza Langari. *Fuzzy logic: intelligence, control, and information*, volume 1. Prentice Hall Press, Upper Saddle River, NJ, 1999.

[YWC$^+$18] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. Taking human out of learning applications: A survey on automated machine learning. pages 1–26, 2018.

[YWW10] Ying Yang, Geoffrey I. Webb, and Xindong Wu. Discretization methods. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook, 2nd ed*, pages 101–116. Springer, 2010.

[Zha94] Kang Zhang. A review of exploitation of and-parallelism and combined and/or-parallelism in logic programs. *ACM SIGPLAN Notices*, 29(2):25–32, 1994.

[Zha19] Dengsheng Zhang. *Wavelet Transform*, pages 35–44. Springer International Publishing, Cham, 2019.

[ZTX93] Yuhua Zheng, Honglei Tu, and Li Xie. And/or parallel execution of logic programs: Exploiting dependent and-parallelism. *ACM SIGPLAN Notices*, 28(5):19–28, 1993.

[ZY04] Anmin Zhu and Simon X. Yang. A fuzzy logic approach to reactive navigation of behavior-based mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA'04)*, volume 5, pages 5045–5050. IEEE, 2004.

[ZYH$^+$18] Hao Zhou, Tom Young, Minlie Huang, Haizhou Zhao, Jingfang Xu, and Xiaoyan Zhu. Commonsense knowledge aware conversation generation with graph attention. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 4623–4629. ijcai.org, 2018.