On the Design of PSyKI: a Platform for Symbolic Knowledge Injection into Sub-Symbolic Predictors

On the Design of PSyKI: A Platform for Symbolic Knowledge Injection into Sub-Symbolic Predictors

Matteo Magnini[1][0000−0001−9990−420X], Giovanni Ciatto[1][0000−0002−1841−8996], and Andrea Omicini[1][0000−0002−6655−3869]

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna
{matteo.magnini, giovanni.ciatto, andrea.omicini}@unibo.it

Abstract. A long-standing ambition in artificial intelligence is to integrate predictors' *inductive* features (i.e., learning from examples) with *deductive* capabilities (i.e., drawing inferences from symbolic knowledge). Many algorithms methods in the literature support injection of symbolic knowledge into predictors, generally following the purpose of attaining better (i.e., more effective or efficient w.r.t. predictive performance) predictors. However, to the best of our knowledge, running implementations of these algorithms are currently either proof of concepts or unavailable in most cases. Moreover, an unified, coherent software framework supporting them as well as their interchange, comparison, and exploitation in arbitrary ML workflows is currently missing. Accordingly, in this paper we present PSyKI, a platform providing general-purpose support to symbolic knowledge injection into predictors via different algorithms.

Keywords: symbolic knowledge injection · explainable AI · XAI · neural networks · PSyKI.

1 Introduction

Within the scope of supervised machine learning (ML), it is a common practice to rely upon *sub-symbolic* predictors such as neural networks (NN) to mine the useful information buried in data. However, given that they do not provide any intelligible representation of what they learn from data, they are considered as black boxes [13]. This is becoming troublesome in many application scenarios (e.g., domains concerning healthcare, law, finance)—the reason being: it is non-trivial to forecast what will black-box predictors actually learn from data, or whether and to what extent they will learn general, reusable information.

State-of-the-art solutions currently address this issue by supporting a plethora of methods for "opening the black-box" [9]—i.e. inspecting or debugging the inner functioning of NN. Along this line, a common strategy is to perform symbolic knowledge extraction (SKE) – producing human-interpretable information – on sub-symbolic predictors. For instance, PSyKE [15] is a technological tool supporting SKE via several algorithms—dual in its intents to what we propose here.

Another strategy to deal with opaque predictors is to prevent them from becoming black boxes. To do so, the training process of a NN is controlled in such a way that the designer suggests what -(prior knowledge) predictor might (not) learn. In other words, the intent is to transfer the designer's common-sense into the predictor. Along this line, we call *symbolic knowledge injection* (SKI) the task of letting a sub-symbolic predictor exploit formal, symbolic information.

Notably, SKI brings a number of key benefits to the training of sub-symbolic predictors, possibly mitigating the issues arising from their opacity. In particular, SKI is expected to: *(i)* prevent predictors from becoming complete black boxes during their training; *(ii)* reduce learning time by immediately providing the knowledge predictors should otherwise struggle to learn by processing huge amount of data; *(iii)* mitigate the issues arising from the lack of sufficient amounts of training data (as under-represented situations can be suitably represented in symbols); *(iv)* improve predictors' predictive performance in corner cases—such as in presence of unbalanced and overlapping classes.

As further discussed in Section 2, virtually all SKI methods proposed in literature share a general workflow (Section 2.3), which can be briefly summarised as: *(i)* identify a suitable predictor w.r.t. the ML task at hand, *(ii)* produce some symbolic knowledge aimed at describing corner cases or notable situations, *(iii)* apply the SKI method that given the given predictor and knowledge, hence generating a new predictor that encapsulates the knowledge, *(iv)* train the new predictor on the available data, as usual. Hence, in principle, SKI methods are interchangeable at the functional level—despite each method may more (or less) adequate to particular classes of ML tasks/problems. However, to the best of our knowledge, running implementations of SKI algorithms are currently either proof of concepts or unavailable in most cases. Moreover, an unified, coherent software framework supporting them all – as well as their interchange, comparison, and exploitation in arbitrary ML workflows – is currently missing.

To mitigate this issue, in this paper we present the design of PSyKI, a Platform for Symbolic Knowledge *Injection*. PSyKI is conceived as an open library where different sorts of knowledge injection algorithms can be realised, exploited, or compared. PSyKI is a tool for data scientists willing to experiment already existing SKI algorithms, or to invent new ones while making them available under a general API. In this sense, PSyKI is complementary w.r.t. PSyKE.

Accordingly, the remainder of this paper is organised as follows. In Section 2 we summarise the background of SKI, eliciting a number of related works. Then in Section 3 we describe the design of PSyKI. Section 4 reports a case study of injection in a well known domain. Finally, conclusion are drawn in Section 5.

2 Knowledge Injection Background

Many methods and techniques for injecting symbolic knowledge into ML predictors have been proposed into the literature. Virtually all of them assume

- knowledge is encoded via some subset of first order logic (FOL),
- while ML predictors consist of neural networks (NN).

Arguably, possible motivations behind these choices are the flexibility of logic (and FOL in particular) in expressing symbolic information, and the malleability and composability of NN—which can the structured in manifold ways, to serve disparate purposes.

However, the many SKI methods from the literature can be categorised into two major groups, depending on *how* they perform the injection of symbolic knowledge into neural networks. Namely, some methods perform injection by *constraining* neural networks' training, while others affect their inner *structure*. Approaches from first group perform injection during the network's training, using the symbolic knowledge as either a *constraint* or a guide for the optimization process. Conversely, approaches of the second sort perform injection by *structuring* the network's architecture to make it mirror the symbolic knowledge.

In the following subsections, we non-exhaustively enumerate and describe some major SKI techniques from the literature, evenly distributed among both groups. A summary of the surveyed approaches is reported in Table 1.

2.1 Constraining Neural Networks

The key idea of SKI techniques based on constraints is to induce a penalty during the training process of the predictor. A cost is applied in some way whenever the network violates the prior knowledge. A common way to do is by interpreting logic formulæ as fuzzy logic functions so that the penalty is proportional to the degree of violation.

Through back propagation, the weights of the NN are optimised to minimise both the prediction error and the additional penalty. In this way the predictor is *constrained* to be compliant to the prior knowledge with a certain extent.

CODL (COnstrained Driven Learning) [4], for instance, is an early work concerning the injection of custom symbolic rules into a hidden Markov model (HMM) for a natural language process (NLP) task. Constraints encode structural information and interdependencies among labels. Then, they are injected in the semi-supervised learning process of the HMM within a distance function between the actual output and the output space that is compliant to the knowledge constraints.

SBR (Semantic Based Regularisation) [5] is a framework for knowledge injection in form of FOL rules into kernel machines (KM) such as support vector machines (SVM). Knowledge is a set of constraints – transformed in fuzzy functions – that must be satisfied in addition to the traditional smoothness regularization term. The semantic inference can be back-propagated down to the kernel machines using any gradient-based schema during training.

A development of SBR is the work presented in [6] that enables the injection of knowledge into NN. SBR is used as underlying framework to represent prior knowledge. Constraints are integrated into the loss function of the NN so that the model minimizes both the error between real and expected value and the cost introduced by the constraints.

SLF (Semantic Loss Function) [18] is a technique to derive a semantic loss function that bridges NN output vectors and logical constraints that are provided in form of simple propositional logic. Experimental results show an improvement of the NN ability to predict structured objects, such as rankings and paths.

Table 1: Summary of the presented SKI techniques.

Algorithm	**Typology**	**Predictor**	**Knowledge**	**Task**
C-IL^{2}P	Structuring	NN	propositional logic	classification
CODL	Contraining	HMM	custom logic rules	classification
Fibring	Structuring	NN	first order logic	mimic LP
GFNN	Hybrid	NN	propositional logic	classification, regression
KBANN	Structuring	NN	propositional logic	classification
SBR	Constraining	KM and NN	first order logic	classification
SLF	Constraining	NN	propositional logic	classification
Student-Teacher	Structuring	NN	first order logic	classification

2.2 Structuring Neural Networks

Differently from the constraining techniques, the idea of SKI based on structuring is to alter the architecture of the predictor to integrate the prior knowledge. In other words, structural components of the predictor are built or modified in such a way that they can mimic to a certain extent the behaviour of the provided knowledge. The structural modification may occur at several levels: *(i)* impose constraints on the values of weights, *(ii)* add additional neurons that resemble the knowledge to be injected, *(iii)* generate the entire NN from logic rules.

One of the earliest works that performing SKI in this way is KBANN [16]. Logic rules with limited expressiveness – propositional, variable-free and not recursive – are successfully embedded in NN. Each entity of the knowledge base is mapped into a neuron and relations between entities into edges.

Conversely, in [17], the authors discuss how simple rule-based logical expressions can be approximated as Gaussian basis functions that represent the certainty of given rules. Outputs of such functions are then weighted together to obtain the output of the NN (we call such networks GFNN), representing the composition of activated rules. This is an example of SKI approach with both constraining and structuring.

C-IL^{2}P system is presented in [8], aiming at integrating inductive learning with deductive learning. The system provides a straightforward translation algorithm from grounded propositional logic programs to neural networks that are then trained over examples.

An interesting student–teacher approach for SKI is presented in [11] and [12]. Authors inject grounded FOL rules into the weights of NN using a two networks training strategy. This is achieved by forcing a student (first network) to emulate the predictions of a rule-regularized teacher (second network) evolving both models iteratively.

Fibred neural networks are introduced in [7] and [2]. Authors propose to represent FOL programs as fibred NN, capable of implementing recursion and enabling them to perform symbolic computation. A fibred NN is a network composed by two (or possibly more) networks, A and B, where the output of i^{th} neuron of network A is given by the output of network B and it is defined a function $\phi_i : I \rightarrow W$ from A to B, I is the input of neuron i^{th} of A and W is the set of weights of B.

2.3 Workflow

All the aforementioned methods focus on specific types of predictors and logic formulæ that can be quite different from each others: however, all methods share the same general workflow, overviewed in Figure 1. Injection algorithms accept both a predictor P and prior symbolic knowledge φ as input, and they generate a new predictor P' as output, which is then trained over data. If knowledge injection is made via constraining and constraints affect the actual prediction value, then they could be removed after the training, as they are not necessary anymore. We underline that the final trained predictor P'' can be re-used to start a new cycle of SKI with possibly different knowledge or injection algorithm.

Symbolic knowledge φ is usually provided as logic formulæ, yet it can have other representations. However, it is unlikely that knowledge in this form cannot be directly injected into a sub-symbolic predictor. It must be first embedded into a machine injectable form. The embedding process is depicted in Figure 2 and it is composed of two operations: parsing (Π) and fuzzification (ζ). Parsing transforms symbolic knowledge into a visitable data structure, for example in the case of logic formulæ into abstract syntax trees (AST). Knowledge in form of visitable data structure φ' can then be processed by a software component – that we call fuzzifier – that create sub-symbolic (injectable) objects φ''. The fuzzification process it is necessary to transform a *crispy* symbolic formula into a *fuzzy* interpretation, in other words transform a Boolean function that outputs a Boolean value into a function (or other sub-symbolic objects) that outputs a continuos value. We do not impose limits on the nature of φ'' that can be functions, sets of structural components (e.g. NN layers, whole NN, etc.), basically anything that is sub-symbolic and therefore can be used in conjunction with P.

Literature is far less rich when software applications that support SKI algorithms or the developing of new ones are concerned. LYRICS [14] is an example of interface for the integration of logic inference and deep learning. The application targets NN as ML predictors and logic rules represented in FOL, and it performs injection via constraining. In the next section we present the design of a more general software inspired by the workflow in Figure 1, so not imposing any constraint on the nature of predictors or the form of symbolic knowledge.

3 PSyKI

PSyKI is a software library that provides support for the injection of prior symbolic knowledge into sub-symbolic predictors by letting the users choose the

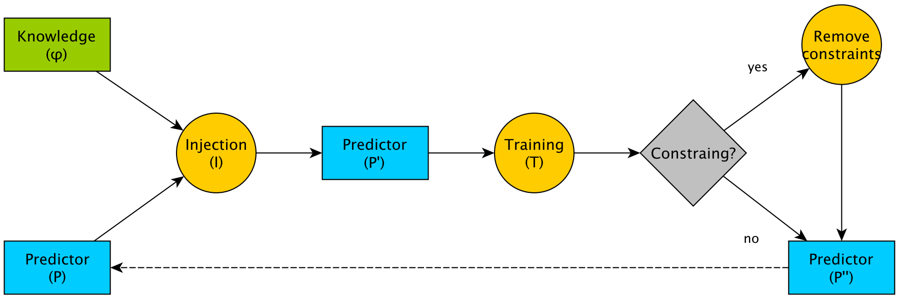


Fig. 1: Common workflow of symbolic knowledge injection.

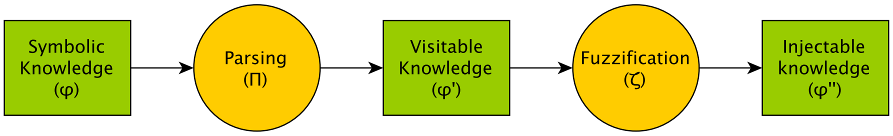


Fig. 2: Symbolic knowledge embedding into sub-symbolic form.

most adequate method with respect to the ML task to accomplish. PSyKI is a tool for data scientists who want to experiment already existing SKI algorithms or who want to invent new ones. Source code of PSyKI is publicly available at `https://github.com/psykei/psyki-python`.

3.1 Architecture

Essentially, PSyKI is designed around the notion of *injector*, whose API is showed in Figure 3. An injector is any algorithm accepting a ML predictor and prior symbolic knowledge – predominantly logic formulæ – as input that produces a new predictor as output. In order to properly perform injection, injectors may require additional information such as algorithm specific hyperparameters. The general workflow for SKI with PSyKI that we propose is compliant to the one presented in Section 2.3.

PSyKI offers support for the processing of the symbolic knowledge represented via logic formulæ. Based on the sort of logic, user can build an abstract syntax tree (AST) for each formula. The AST can be inspected through a *fuzzifier* via pattern visitor to encode the symbolic knowledge to a sub-symbolic form (e.g. fuzzy logic functions, ad-hoc layers). The resulting sub-symbolic object can finally be used by an injector to create a new predictor. This process – denoted with ζ in Figure 2 – is injector specific; instead, the same parser Π can be used for logic formulæ of the same sort independently of the injector.

The software is organized into well-separated packages to ensure easy extensibility towards new sort of logic and fuzzifiers—see Figure 4. AST is a *formula*

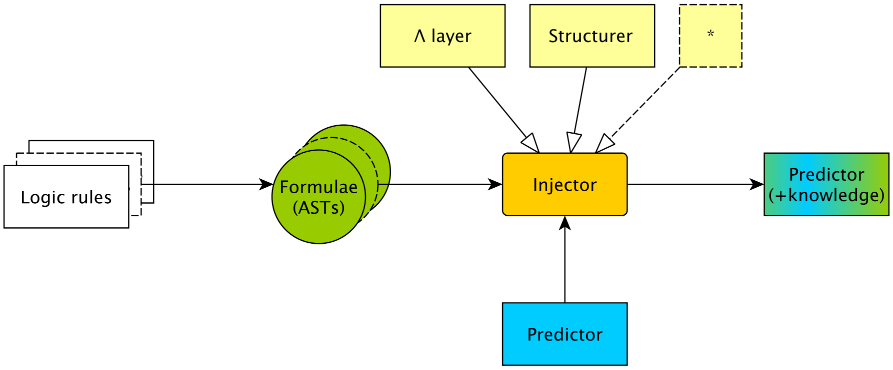


Fig. 3: PSyKI design.

object and obviously it can have different language specific elements w.r.t. the logic form that is covered. Each *formula* implementation is self-contained inside a stand alone package so that if a user wants to add a new logic form it is sufficient to add its implementation in a new package. Similarly, a *fuzzifier* object that targets a specific logic form can be add inside the same package of the logic, there can be any number of fuzzifiers for a given logic.

3.2 Knowledge Parsing

A crucial point in the SKI workflow is the embedding of knowledge from symbolic into sub-symbolic form. Ideally, there is no constraint on the formalism used to represent the prior knowledge (e.g. logic formulæ, knowledge graph, etc.).

The most common knowledge representation form that SKI algorithms claim to support is FOL (or subsets of FOL). However, there are characteristics of FOL that are not ideal for some predictors. Recursion and function symbols – that allow recursive structures – cannot be easily integrated into a predictor that is acyclic (no recursive) by construction such as conventional NN (virtually all NN, with few exceptions like fibred [2]). Conversely, we consider one of the most general and expressive logic formalism that does not support recursion and function symbols: stratified Datalog with negation.

Stratified Datalog logic formulæ with negation is a subset of FOL without recursive clause definition where knowledge is represented via function-free Horn clauses [1]. Horn clauses, in turn, are formulæ of the form $\phi \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots$ denoting a logic implication ($\leftarrow$) stating that ϕ (the head of the clause) is implied by the conjunction among a number of atoms $\psi_1, \psi_2, \ldots$ (the body of the clause). Since we rely on Datalog *with negation*, we allow atoms in the bodies of clauses to be negated. In case the i^{th} atom in the body of some clause is negated, we write $\neg\psi_i$. There, each atom $\phi, \psi_1, \psi_2, \ldots$ may be a predicate of arbitrary arity.

An l-ary predicate p denotes a relation among l entities: $p(t_1, \ldots, t_l)$ where each t_i is a term, i.e. either a constant (denoted in `monospace`) representing a

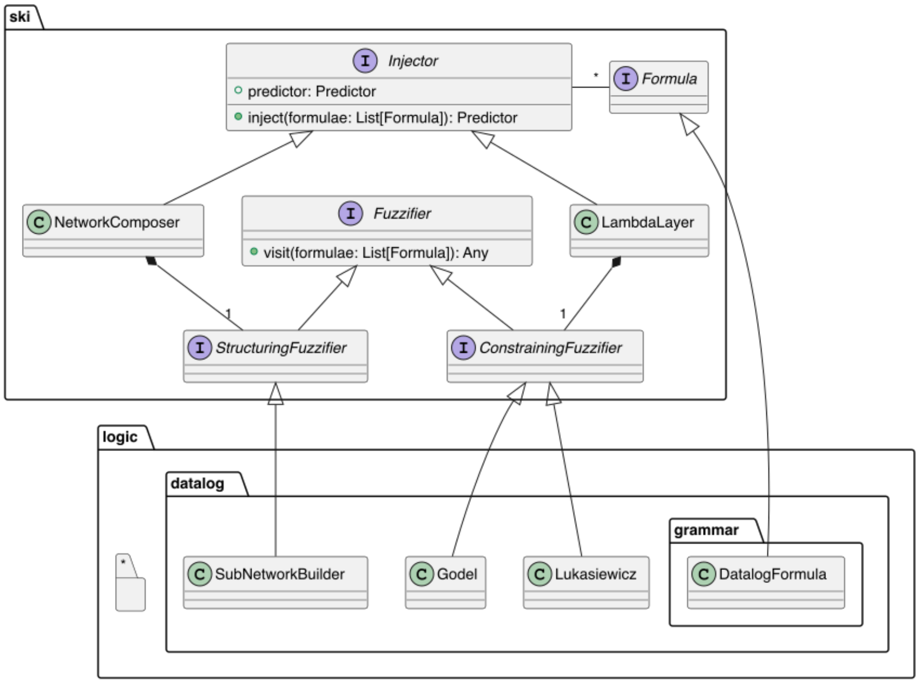


Fig. 4: Class diagram of PSyKI. Main entities are *Injector*, *Formula*, and *Fuzzifier*. Package *logic.datalog* is an exemplification showing two *Injector* implementations and their relationships.

particular entity, or a logic variable (denoted by *Capitalised Italic*) representing some unknown entity or value. Well-known binary predicates are admissible too, such as $>$, $<$, $=$, etc., which retain their usual semantics from arithmetic. For the sake of readability, we may write these predicates in infix form—hence $>(X, 1) \equiv X > 1$.

Consider for instance the case of a rule aimed at defining when a Poker hand can be classified as a pair—the example may be useful in the remainder of this paper. Assuming that a Poker hand consists of 5 cards, each one denoted by a couple of variables R_i, S_i – where R_i (resp. S_i) is the rank (resp. seed) of the i^{th} card in the hand –, hands of type *pair* may be described via a set of clauses such as the following one:

$$\begin{array}{r}
pair(R_1, S_1, \ldots, R_5, S_5) \leftarrow R_1 = R_2 \\
pair(R_1, S_1, \ldots, R_5, S_5) \leftarrow R_2 = R_3 \\
\vdots \\
pair(R_1, S_1, \ldots, R_5, S_5) \leftarrow R_4 = R_5
\end{array}$$

To support injection into a particular predictor, we further assume the input knowledge base defines at least one outer relation – say *output* or *class* – involving as many variables as the input and output features the predictor has been trained upon. Such a relation may be defined via one or more clauses, and each clause may possibly leverage on other predicates in their bodies. In turn, each predicate may be defined through one or more clause. In that case, since we rely on *stratified* Datalog, we require the input knowledge to *not* include any (directly or indirectly) *recursive* clause definition.

For example, for a 3-class classification task, any provided knowledge base should include a clause such as the following one:

$$\begin{array}{r}
class(\bar{X}, \mathtt{y}_1) \leftarrow p_1(\bar{X}) \wedge p_2(\bar{X}) \\
class(\bar{X}, \mathtt{y}_2) \leftarrow p_1'(\bar{X}) \wedge p_2'(\bar{X}) \\
class(\bar{X}, \mathtt{y}_3) \leftarrow p_1''(\bar{X}) \wedge p_2''(\bar{X})
\end{array}$$

where $\bar{X}$ is a tuple having as many variables as the neurons in the output layer, and $\mathtt{y}_i$ is a constant denoting the i^{th} class.

Once that the logic has been formalized, the implementation of a *Formula* – visitable data structure like an AST – is quite straight forward, see Figure 5. It is convenient to define a custom adapter from a generated AST by a third part library – such as ANTLR – to *Formula* object. Knowledge in *Formula* form can be embedded with a fuzzifier into a sub-symbolic form and finally injected into a predictor.

3.3 Knowledge Fuzzification

In this section we propose two possible fuzzifications of *Formula* objects suitable for injection via constraining and structuring.

In the former method – which we call *Lukasiewicz* – each *Formula* is converted into a real-valued function aimed at computing the cost of violating that

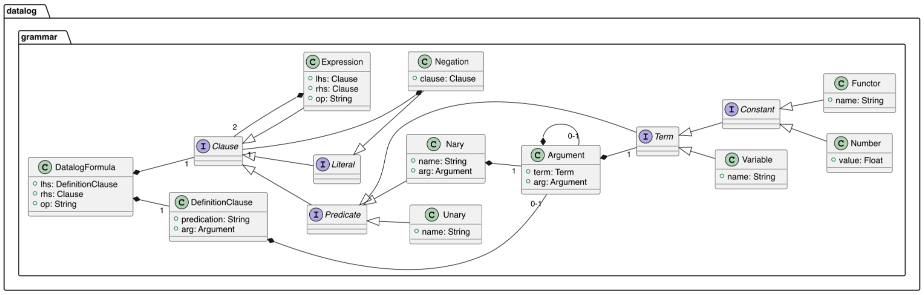


Fig. 5: Class diagram of the *Formula* implementation for Datalog logic formulæ.

formula. To serve this purpose, we rely on a multi-valued interpretation of logic inspired to Lukasiewicz's logic [10]. Accordingly, we encode each formula via the $[\![\cdot]\!]$ function, mapping logic formulæ into real-valued functions accepting real vectors of size $m+n$ as input and returning scalars in $\mathbb{R}$ as output. These scalars are then clipped into the $[0, 1]$ range, via the $\eta : \mathbb{R} \to [0, 1]$, defined as follows:

$$\eta(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } 0 < x < 1 \\ 1 & \text{if } x \geq 1 \end{cases} \tag{1}$$

Hence, the penalty associated with the i^{th} predictor's output violating rule ϕ_i can be written as $c_i(\mathbf{x}, y_i) = \eta([\![\phi_i]\!](\mathbf{x}, y_i))$. The $[\![\cdot]\!]$ encoding function is recursively defined in Table 2. Such functions can be integrated into a predictor to constraint its training — it is unlikely that P'' makes predictions that violate φ.

The second method – namely *SubNetworkBuilder* – transforms each *Formula* into a neural network (modules). Each modules is responsible to mimic the behaviour of a single logic formula, more precisely to evaluate the truth degree of the right hand side of the formula. To do so, all logic operators of the formulæ have to be mapped into an ad-hoc layer that computes the associated fuzzy function for the specific operator (like in the previous fuzzification method, but values are not necessary intended to be penalties). Layers are then recursively combined following the AST to build a NN. Finally, modules can be permanently integrated inside a NN predictor able to exploit both prior knowledge and induction by examples.

4 Case Study

In the following part of this manuscript we present how to effectively use PSyKI in a ML task. We choose to apply knowledge injection in a domain where it is easy to express correct logic rules and at the same time it is difficult for sub-symbolic predictors to achieve optimal results.

Table 2: Two logic formulæ's encodings into real-valued functions (the first with 0 representing true and 1 false, vice versa in the second). There, X is a logic variable, while x is the corresponding real-valued variable, whereas is $\bar{X}$ a tuple of logic variables. Similarly, $\mathtt{k}$ is a numeric constant, and k is the corresponding real value, whereas $\mathtt{k}_i$ is the constant denoting the i^{th} class of a classification problem. Finally, $\mathrm{expr}(\bar{X})$ is an arithmetic expression involving the variables in $\bar{X}$.

Formula	**Continuos interpretation 1**	**Continuos interpretation 2**
$\llbracket \neg\phi \rrbracket$	$\eta\{1 - \llbracket\phi\rrbracket\}$	$\eta\{1 - \llbracket\phi\rrbracket\}$
$\llbracket \phi \wedge \psi \rrbracket$	$\eta\{max\{\llbracket\phi\rrbracket, \llbracket\psi\rrbracket\}\}$	$\eta\{min\{\llbracket\phi\rrbracket, \llbracket\psi\rrbracket\}\}$
$\llbracket \phi \vee \psi \rrbracket$	$\eta\{min\{\llbracket\phi\rrbracket, \llbracket\psi\rrbracket\}\}$	$\eta\{max\{\llbracket\phi\rrbracket, \llbracket\psi\rrbracket\}\}$
$\llbracket \phi = \psi \rrbracket$	$\eta\{\lvert\llbracket\phi\rrbracket - \llbracket\psi\rrbracket\rvert\}$	$\eta\{\llbracket\neg(\phi \neq \psi)\rrbracket\}$
$\llbracket \phi \neq \psi \rrbracket$	$\llbracket\neg(\phi = \psi)\rrbracket$	$\eta\{\lvert\llbracket\phi\rrbracket - \llbracket\psi\rrbracket\rvert\}$
$\llbracket \phi > \psi \rrbracket$	$\eta\{1 - max\{0, \llbracket\phi\rrbracket - \llbracket\psi\rrbracket\}\}$	$\eta\{max\{0, \llbracket\phi\rrbracket - \llbracket\psi\rrbracket\}\}$
$\llbracket \phi \geq \psi \rrbracket$	$\llbracket(\phi > \psi) \vee (\phi = \psi)\rrbracket$	$\eta\{\llbracket(\phi > \psi) \vee (\phi = \psi)\rrbracket\}$
$\llbracket \phi < \psi \rrbracket$	$\llbracket\neg(\phi \geq \psi)\rrbracket$	$\eta\{max\{0, \llbracket\psi\rrbracket - \llbracket\phi\rrbracket\}\}$
$\llbracket \phi \leq \psi \rrbracket$	$\llbracket\neg(\phi > \psi)\rrbracket$	$\eta\{\llbracket(\phi < \psi) \vee (\phi = \psi)\rrbracket\}$
$\llbracket \phi \Rightarrow \psi \rrbracket$	$\eta\{max\{0, \llbracket\psi\rrbracket - \llbracket\phi\rrbracket\}\}$	$\eta\{min\{1, 1 - \llbracket\psi\rrbracket + \llbracket\phi\rrbracket\}\}$
$\llbracket \phi \Leftarrow \psi \rrbracket$	$\eta\{max\{0, \llbracket\phi\rrbracket - \llbracket\psi\rrbracket\}\}$	$\eta\{min\{1, 1 - \llbracket\phi\rrbracket + \llbracket\psi\rrbracket\}\}$
$\llbracket \phi \Leftrightarrow \psi \rrbracket$	$\eta\{max\{0, \lvert\llbracket\phi\rrbracket - \llbracket\psi\rrbracket\rvert\}\}$	$\eta\{min\{1, 1 - \lvert\llbracket\phi\rrbracket - \llbracket\psi\rrbracket\rvert\}\}$
$\llbracket \mathrm{expr}(\bar{X}) \rrbracket$	$\mathrm{expr}(\llbracket\bar{X}\rrbracket)$	$\mathrm{expr}(\llbracket\bar{X}\rrbracket)$
$\llbracket \mathtt{true} \rrbracket$	0	1
$\llbracket \mathtt{false} \rrbracket$	1	0
$\llbracket X \rrbracket$	x	x
$\llbracket \mathtt{k} \rrbracket$	k	k
$\llbracket p(\bar{X}) \rrbracket^{**}$	$\llbracket\psi_1 \vee \ldots \vee \psi_k\rrbracket$	$\llbracket\psi_1 \vee \ldots \vee \psi_k\rrbracket$
$\llbracket class(\bar{X}, \mathtt{y}_i) \leftarrow \psi \rrbracket$	$\llbracket\psi\rrbracket^{*}$	$\llbracket\psi\rrbracket^{*}$

* encodes the penalty for the i^{th} output

** assuming predicate p is defined by k clauses of the form:
$p(\bar{X}) \leftarrow \psi_1, \ \ldots, \ p(\bar{X}) \leftarrow \psi_k$

Poker hand dataset [3] subtends a multi-classification task on a finite – yet very large – discrete domain, where classes are overlapped and heavily unbalanced, while exact classification rules can be written in logic formulæ. It consists of a tabular dataset, containing 1,025,010 records—each one composed by 11 features. Each record encodes a poker hand of 5 cards. Hence, each records involves 5 couples of features – denoting the cards in the hand –, plus a single categorical feature denoting the class of the hand. Two features are necessary to identify each card: suit and rank. Suit is a categorical feature (heart, spade, diamond and club), while rank is ordinal feature—suitably represented by an integer between 1 and 13 (ace to king). The multi-classification task consists in predicting the poker hand's class. Each hand may be classified as one of 10 different classes denoting the nature of the hand according to the rules of Poker (e.g., nothing, pair, double pair, flush). We use 25,010 records for training and the remaining million for testing, as shown in Table 3.

Table 3: Poker hand dataset statistics, per class.

Class	**Train. Instances**	**Train. Freq. (%)**	**Test Instances**	**Test Freq. (%)**
nothing	12,493	49.95	501,209	50.12
pair	10,599	42.38	422,498	42.25
two pairs	1,206	4.82	47,622	4.76
three of a kind	513	2.05	21,121	2.11
straight	93	0.37	3,885	0.39
flush	54	0.22	1,996	0.2
full house	36	0.14	1,424	0.14
four of a kind	6	0.024	230	0.023
straight flush	5	0.02	12	0.001
royal flush	5	0.02	3	$1.2 \cdot 10^{-5}$
Total	25,010	100	1,000,000	100

We define a *class* rule for each class, encoding the preferred way of classifying a Poker hand. For example, let $\{S_1, R_1, \ldots, S_5, R_5\}$ be the logic variables representing a Poker hand (S for suit and R for rank), then for class `flush` we define the following rule:

$$\begin{aligned} &\mathit{class}(R_1, S_1, \ldots, R_5, S_5, \texttt{flush}) \leftarrow \mathit{flush}(R_1, S_1, \ldots, R_5, S_5) \\ &\mathit{flush}(R_1, S_1, \ldots, R_5, S_5) \leftarrow S_1 = S_2 \wedge S_1 = S_3 \wedge S_1 = S_4 \wedge S_1 = S_5 \end{aligned} \tag{2}$$

All other rules have the same structure as equation 2: the left-hand side declares the expected class, while the right-hand side describes the necessary conditions for that class—possibly, via some ancillary predicates such as *flush*. Table 4 provides an overview of all the rules we rely upon in our experiments.

Class	Logic Formulation
Pair	$\mathit{class}(R_1, \ldots, S_5, \texttt{pair}) \leftarrow \mathit{pair}(R_1, \ldots, S_5)$ $\mathit{pair}(R_1, \ldots, S_5) \leftarrow R_1 = R_2$ $\vdots$ $\mathit{pair}(R_1, \ldots, S_5) \leftarrow R_4 = R_5$
Two of a Kind	$\mathit{class}(R_1, \ldots, S_5, \texttt{two}) \leftarrow \mathit{two}(R_1, \ldots, S_5)$ $\mathit{two}(R_1, \ldots, S_5) \leftarrow R_1 = R_2 \wedge R_3 = R_4$ $\vdots$ $\mathit{two}(R_1, \ldots, S_5) \leftarrow R_2 = R_3 \wedge R_4 = R_5$
Three of a Kind	$\mathit{class}(R_1, \ldots, S_5, \texttt{three}) \leftarrow \mathit{three}(R_1, \ldots, S_5)$ $\mathit{three}(R_1, \ldots, S_5) \leftarrow R_1 = R_2 \wedge R_2 = R_3$ $\vdots$ $\mathit{three}(R_1, \ldots, S_5) \leftarrow R_3 = R_4 \wedge R_4 = R_5$
Straight	$\mathit{class}(R_1, \ldots, S_5, \texttt{straight}) \leftarrow \mathit{royal}(R_1, \ldots, S_5)$ $\mathit{class}(R_1, \ldots, S_5, \texttt{straight}) \leftarrow \mathit{straight}(R_1, \ldots, S_5)$ $\mathit{straight}(R_1, \ldots, S_5) \leftarrow (R_1 + R_2 + R_3 + R_4 + R_5) = (5 \cdot \min(R_1, \ldots, R_5) + 10) \wedge \neg \mathit{pair}(R_1, \ldots, S_5)$ $\mathit{royal}(R_1, \ldots, S_5) \leftarrow \min(R_1, \ldots, R_5) = 1 \wedge (R_1 + R_2 + R_3 + R_4 + R_5 = 47) \wedge \neg \mathit{pair}(R_1, \ldots, S_5)$
Flush	$\mathit{class}(R_1, \ldots, S_5, \texttt{flush}) \leftarrow \mathit{flush}(R_1, \ldots, S_5)$ $\mathit{flush}(R_1, \ldots, S_5) \leftarrow S_1 = S_2 \wedge S_1 = S_3 \wedge S_1 = S_4 \wedge S_1 = S_5$
Four of a Kind	$\mathit{class}(R_1, \ldots, S_5, \texttt{four}) \leftarrow \mathit{four}(R_1, \ldots, S_5)$ $\mathit{four}(R_1, \ldots, S_5) \leftarrow R_1 = R_2 \wedge R_2 = R_3 \wedge R_3 = R_4$ $\vdots$ $\mathit{four}(R_1, \ldots, S_5) \leftarrow R_2 = R_3 \wedge R_3 = R_4 \wedge R_4 = R_5$
Full House	$\mathit{class}(R_1, \ldots, S_5, \texttt{full}) \leftarrow \mathit{three}(S_1, \ldots, R_5) \wedge \mathit{two}(S_1, \ldots, R_5) \wedge \neg four(S_1, \ldots, R_5)$
Straight Flush	$\mathit{class}(R_1, \ldots, S_5, \texttt{straight\_flush}) \leftarrow \mathit{straight}(R_1, \ldots, S_5) \wedge \mathit{flush}(R_1, \ldots, S_5)$ $\mathit{class}(R_1, \ldots, S_5, \texttt{straight\_flush}) \leftarrow \mathit{royal}(R_1, \ldots, S_5) \wedge \mathit{flush}(R_1, \ldots, S_5)$
Royal Flush	$\mathit{class}(R_1, \ldots, S_5, \texttt{royal}) \leftarrow \mathit{royal}(R_1, \ldots, S_5) \wedge \mathit{flush}(R_1, \ldots, S_5)$
Nothing	$\mathit{class}(R_1, \ldots, S_5, \texttt{nothing}) \leftarrow \neg \mathit{pair}(R_1, \ldots, S_5) \wedge \neg \mathit{flush}(R_1, \ldots, S_5) \wedge \neg \mathit{straight}(R_1, \ldots, S_5) \wedge \neg \mathit{royal}(R_1, \ldots, S_5)$

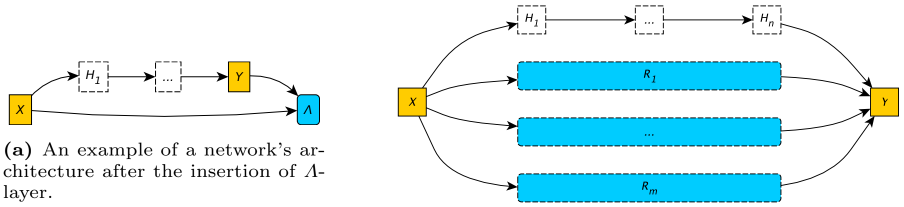
Table 4: Datalog formulæ describing poker hands.

4.1 *Λ*Layer

ΛLayer is a constraining injection technique that targets NN of any shape. It works by appending one further layer – the *Λ-layer* henceforth – at the output end of the neural network – see Figure 6a –, and by training the overall network as usual, via gradient descent or others strategies. The *Λ*-layer is in charge of introducing an error (w.r.t. the actual prediction provided by the network's output layer) whenever the prediction violates the symbolic knowledge. The error is expected to affect the gradient descent – or whatever optimisation function – in such a way that violating the symbolic knowledge is discouraged. Once the NN training is over, the injection phase is considered over as well, hence the *Λ*-layer can be removed and the remaining network can be used as usual. Hence, no architectural property of the original network is hindered by the addition of the *Λ*-layer. The error computed in the *Λ*-layer is a fuzzy function derived from the original symbolic knowledge. In this case, we use *Lukasiewicz* fuzzifier with the first continuos interpretation of Table 2.

4.2 NetworkComposer

NetworkComposer is a straightforward structuring injection method that targets NN of any shape. A neural network architecture is extended with additional neural *modules*, structured to reflect and mimic the symbolic knowledge provided by designers. Here, we choose to encode the symbolic knowledge with *SubNetworkBuilder* fuzzifier using the second continuos interpretation presented in Ta-



(a) An example of a network's architecture after the insertion of Λ-layer.

(b) An example of a network's architecture after the insertion of *modules*.

Fig. 6: Resulting predictors' architectures after the injection of prior knowledge.

ble 2. There, a module is a sub-network having the same input layer and output layers of the original network. Figure 6b shows the general architecture of the resulting NN after the injection of m modules (represented as blue rectangles), corresponding to as many rules to be injected. These modules can be arbitrarily complex sub-networks, sharing the same input and output layers of the original NN. White boxes represent arbitrary hidden layers $H_1, H_2, \ldots$ of the original NN, whereas X is the input layer and Y is the output layer.

4.3 Results

We conduct SKI into a 3-layers fully-connected NN with random weights initialization. The predictor is always the same – all identical hyper parameters – except for the knowledge injection part. The first and second layers have 64 neurons each, the output layer has 10 neurons, one for each class. Neurons' activation functions is the rectified linear unit, except for the neurons of the last layer that have softmax. During training we choose Adam as optimiser, sparse categorical crossentropy as loss function, and 32 as batch size. In total, for each experiment we train predictors for 100 epochs.

Table 5: Test set accuracy, macro F1-score, and single class accuracies for all different symbolic knowledge injection methods.

SKI	**Accuracy**	**macro F1**	**nothing**	**pair**	**two p.**	**three**	**straight**	**flush**	**full**	**four**	**straight f.**	**royal f.**
No injection	0.966	0.436	0.994	0.966	0.848	0.871	0.159	0.012	0.248	0.052	0	0
ΛLayer	0.989	0.478	0.998	0.999	0.945	0.913	0.501	0.002	0.202	0.03	0	0
NetworkComposer	0.986	0.581	0.998	0.996	0.867	0.9	0.825	0.798	0.195	0.03	0.083	0

Results are reported in Table 5. We notice that the "classic" NN has high accuracy values only for frequent classes, instead it cannot correctly label poker hands for rare classes. Using the constraining *ΛLayer* method, the predictor has general higher single-class accuracies, but for very under-represented classes it continues to fail. This is a common limitation of all SKI methods of this kind, because the knowledge is learnt by the predictor through examples — if there are too few examples, or, no examples at all, then it is impossible for the predictor to learn. The structuring injector *NetworkComposer* has much higher single-class accuracies and F1-measure. This is quite expected because this kind of injector are less dependant to class frequency — prior knowledge is directly encoded inside predictor's structure, not learnt only during the training.

5 Conclusion

In this paper we present the design of PSyKI, a platform for symbolic knowledge injection into sub-symbolic predictors. PSyKI allows users to follow the general knowledge injection workflow common to virtually all SKI methods described in Section 2.3. Practically, PSyKI offers SKI algorithms, and its extendability does not require much effort. It is quite easy to create new algorithms and new knowledge embedding methods. We provide a demonstration of PSyKI in Section 4 using two different injectors in a ML classification task.

In the future we plan to enrich PSyKI with current state-of-the-art SKI algorithms, comparison metrics between the implemented procedures and other utilities—i.e. support for different logics.

Acknowledgments

This paper is partially supported by the CHIST-ERA IV project CHIST-ERA-19-XAI-005, co-funded by EU and the Italian MUR (Ministry for University and Research).

References

1. Ajtai, M., Gurevich, Y.: Datalog vs first-order logic. J. Comput. Syst. Sci. **49**(3), 562–588 (1994). https://doi.org/10.1016/S0022-0000(05)80071-6
2. Bader, S., d'Avila Garcez, A.S., Hitzler, P.: Computing first-order logic programs by fibring artificial neural networks. In: Russell, I., Markov, Z. (eds.) Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA. pp. 314–319. AAAI Press (2005), http://www.aaai.org/Library/FLAIRS/2005/flairs05-052.php
3. Cattral, R., Oppacher, F.: Poker hand data set, UCI machine learning repository (2007), https://archive.ics.uci.edu/ml/datasets/Poker+Hand
4. Chang, M., Ratinov, L., Roth, D.: Guiding semi-supervision with constraint-driven learning. In: Carroll, J.A., van den Bosch, A., Zaenen, A. (eds.) ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30, 2007, Prague, Czech Republic. The Association for Computational Linguistics (2007), https://aclanthology.org/P07-1036/
5. Diligenti, M., Gori, M., Saccà, C.: Semantic-based regularization for learning and inference. Artif. Intell. **244**, 143–165 (2017). https://doi.org/10.1016/j.artint.2015.08.011

6. Diligenti, M., Roychowdhury, S., Gori, M.: Integrating prior knowledge into deep learning. In: Chen, X., Luo, B., Luo, F., Palade, V., Wani, M.A. (eds.) 16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, Cancun, Mexico, December 18-21, 2017. pp. 920–923. IEEE (2017). https://doi.org/10.1109/ICMLA.2017.00-37
7. d'Avila Garcez, A.S., Gabbay, D.M.: Fibring neural networks. In: McGuinness, D.L., Ferguson, G. (eds.) Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA. pp. 342–347. AAAI Press / The MIT Press (2004), http://www.aaai.org/Library/AAAI/2004/aaai04-055.php
8. d'Avila Garcez, A.S., Zaverucha, G.: The connectionist inductive learning and logic programming system. Appl. Intell. **11**(1), 59–77 (1999). https://doi.org/10.1023/A:1008328630915
9. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. **51**(5), 93:1–93:42 (2019). https://doi.org/10.1145/3236009
10. Hay, L.S.: Axiomatization of the infinite-valued predicate calculus. The Journal of Symbolic Logic **28**(1), 77–86 (1963), http://www.jstor.org/stable/2271339
11. Hu, Z., Ma, X., Liu, Z., Hovy, E.H., Xing, E.P.: Harnessing deep neural networks with logic rules. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics (2016). https://doi.org/10.18653/v1/p16-1228
12. Hu, Z., Yang, Z., Salakhutdinov, R., Xing, E.P.: Deep neural networks with massive learned knowledge. In: Su, J., Carreras, X., Duh, K. (eds.) Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016. pp. 1670–1679. The Association for Computational Linguistics (2016). https://doi.org/10.18653/v1/d16-1173
13. Lipton, Z.C.: The mythos of model interpretability. Commun. ACM **61**(10), 36–43 (2018). https://doi.org/10.1145/3233231
14. Marra, G., Giannini, F., Diligenti, M., Gori, M.: LYRICS: A general interface layer to integrate logic inference and deep learning. In: Brefeld, U., Fromont, É., Hotho, A., Knobbe, A.J., Maathuis, M.H., Robardet, C. (eds.) Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11907, pp. 283–298. Springer (2019). https://doi.org/10.1007/978-3-030-46147-8_17
15. Sabbatini, F., Ciatto, G., Calegari, R., Omicini, A.: On the design of psyke: A platform for symbolic knowledge extraction. In: Calegari, R., Ciatto, G., Denti, E., Omicini, A., Sartor, G. (eds.) Proceedings of the 22nd Workshop "From Objects to Agents", Bologna, Italy, September 1-3, 2021. CEUR Workshop Proceedings, vol. 2963, pp. 29–48. CEUR-WS.org (2021), http://ceur-ws.org/Vol-2963/./paper14.pdf
16. Towell, G.G., Shavlik, J.W., Noordewier, M.O.: Refinement ofapproximate domain theories by knowledge-based neural networks. In: Shrobe, H.E., Dietterich, T.G., Swartout, W.R. (eds.) Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes. pp. 861–866. AAAI Press / The MIT Press (1990), http://www.aaai.org/Library/AAAI/1990/aaai90-129.php

17. Tresp, V., Hollatz, J., Ahmad, S.: Network structuring and training using rule-based knowledge. In: Hanson, S.J., Cowan, J.D., Giles, C.L. (eds.) Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]. pp. 871–878. Morgan Kaufmann (1992), http://papers.nips.cc/paper/638-network-structuring-and-training-using-rule-based-knowledge

18. Xu, J., Zhang, Z., Friedman, T., Liang, Y., den Broeck, G.V.: A semantic loss function for deep learning with symbolic knowledge. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5498–5507. PMLR (2018), http://proceedings.mlr.press/v80/xu18h.html