Logic Programming library for Machine Learning: API design and prototype

Giovanni Ciatto[1], Matteo Castigliò[3] and Roberta Calegari[2]

[1]Departement of Computer Science and Engineering (DISI), Alma Mater Studiorum—Università di Bologna
[2]Alma Mater Research Institute for Human Centered AI (AlmaAI), Alma Mater Studiorum—Università di Bologna
[3]Master Student at DISI

Abstract
In this paper we address the problem of hybridising symbolic and sub-symbolic approaches in artificial intelligence, following the purpose of creating flexible and data-driven systems, which are simultaneously comprehensible and capable of automated learning. In particular, we propose a logic API for supervised machine learning, enabling logic programmers to exploit neural networks – among the others – in their programs. Accordingly, we discuss the design and architecture of a library reifying APIs for the Prolog language in the 2P-Kt logic ecosystem. Finally, we discuss a number of snippets aimed at exemplifying the major benefits of our approach when designing hybrid systems.

Keywords
Logic programming, Machine Learning, API, 2P-Kt

1. Introduction

Symbolic and sub-symbolic artificial intelligence (AI) are complementary under several perspectives [1, 2]. For this reason, many recent contributions from the literature are discussing the possible frameworks for their integration and hybridisation [3, 4, 5, 6]. 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. Conversely, 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 paper, we address the issue of supporting machine learning (ML) – and, in particular, neural-networks (NN) based training and inference – in logic programming (LP). We do so by designing and prototyping a logic based API for machine learning. Along this line, our contribution is twofold: *(i)* we let logic programmers exploit the benefits of sub-symbolic AI, and, in particular, neural networks; and *(ii)* we enable the practical experimentation of hybrid

CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy
*Corresponding author.
giovanni.citatto@unibo.it (G. Ciatto); matteo.castiglio@studio.unibo.it (M. Castigliò); roberta.calegari@unibo.it (R. Calegari)
https://about.me/gciatto (G. Ciatto); http://robertacalegari.apice.unibo.it (R. Calegari)
0000-0002-1841-8996 (G. Ciatto); 0000-0003-3794-2942 (R. Calegari)

systems—involving both logic and neural processing of data.

Our logic-based API for ML 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 that are commonly involved in an 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 [7], the JVM platform, and the Prolog language [8]. DeepLearning4J [9] is the underlying library we leverage on in this paper. However, our design is general enough to support other libraries and, possibly, different platforms—e.g. Tensorflow [10] over Python.

Arguably, our work represents the first step towards a wider degree of interoperability among symbolic- 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 LP) and intuitive (via NN) reasoning capabilities. Along this path, the proposed API is a 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 a tool for endowing sub-symbolic predictors with prior, high-level knowledge.

2. Logic library for ML: goals

To properly design a logic library for dealing with hybrid reasoning, some basic goals to achieve should be taken into account: namely, *(i)* enable hybrid reasoning, *(ii)* full support of declarative ML, *(iii)* enable the exploitation of symbolic data sources (in addition to the others), *(iv)* make it possible to select a model via resolution. It is worth mentioning that, each one of these goals comes along with some of the benefits of hybridization, discussed in detail in [2]. In the following details about these goals are discussed.

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.

Declarative ML. Declarative ML is a paradigm by which data scientists' code should only specify *what* an 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 (e.g. Python) and libraries of elementary components to be composed (e.g. Scikit-Learn [11]). However, the solutions proposed so far do not leverage inherently declarative frameworks like LP, but rather object-oriented languages—requiring imperative statements. Hence, to support the declarative expression of an ML workflow in the LP framework, a new logic API is required.

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.

Model selection via resolution. Logic resolution essentially consists 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.

3. ML: key aspects to be supported

To support the aforementioned goals, logic APIs must cover the full gamma of aspects involved in any possible ML workflow, detailed and discussed in this section.

Briefly speaking, an 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. Each ML workflow can be conceived as composed of six major phases – elicited in section 3.1 –, each one involving a number of activities—elicited in section 3.2. Enumerating and defining all possible phases and activities is of paramount importance, as any API for ML should support them all.

3.1. ML phases

From a coarse-grained perspective, a machine learning workflow is composed of six major phases, detailed in the following.

Dataset loading. The first step of any ML workflow consists of loading that dataset in memory for later processing. To support such a step, ML frameworks come with ad-hoc functionalities aimed at loading the dataset by reading a file from the local file system, fetching it from the Web, or 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. Accordingly, the 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. Most common bulk operations of pre-processing are: *(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. In particular, the purpose *(v)* is of paramount importance, as it supports the *test set separation* as well as splitting input-related columns from output-related ones—fundamental operation to enable validation and testing and to support training respectively.

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. A preliminary phase to select the best predictor is a common phase in virtually any ML workflow. Once a particular sort of predictor has been chosen, a way to specify the shape the to-be-trained predictor should have is required. 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. Finally, *hyper-parameters* of the selected algorithm need to be tuned.

Accordingly, the 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 predictor. In case hyper-parameters cannot be conveniently represented as raw logic types (numbers or strings), ad-hoc predicates should be provided as well for constructing structured hyper-parameters values.

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 a 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 the 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.

Inference. Inference 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 training set. 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 follows the training and precedes the 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 this end, 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, the logic API for ML should provide predicates to compute each possible measure.

3.2. ML activities, per phase

Here we elicit the many activities involved in each phase of any ML workflow, and we describe them from a computational perspective—i.e. in terms of the sorts of entities (a.k.a. data types) they accept as input or produce as output (manipulate, for short).

Entities. We start our discussion by identifying the five major sorts of entities each activity may manipulate.

- *Value:* a scalar, vectorial, matrix, or tensorial datum from a given domain (e.g. integer or real numbers, or 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.
- *Dataset:* a collection of values matching a particular schema (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)

Any logic-based API for ML should support the representation, combination, and manipulation of entities of these kinds.

Activities. Each phase of the ML workflow is then characterised by a specific set of activities possibly manipulating entities of any of these sorts. A logic-based API for ML should support them as well. Accordingly, in the remainder of this section, we describe such activities along with the entities they operate upon. In doing so, we partition activities w.r.t. the ML phase they are most commonly exploited into.

Dataset loading. The main activities to support the loading of a dataset into a solver's memory, and its preparation for subsequent processing are

- *Dataset loading:* operation of loading a dataset from either a value – representing either a local or remote file –, or from a Prolog theory
- *Schema declaration:* operation of constructing a representation for a given schema
- *Target features declaration:* operation of tagging a portion of the features of some schema as either inputs or outputs (a.k.a. targets)
- *Dataset splitting:* operation of horizontally partitioning a dataset into two or more smaller datasets.

Dataset pre-processing. Here, they may be willing to define transformations or cascades of transformations (pipelines, henceforth) to be eventually applied to datasets:

- *Transformation declaration:* operation of declaratively encoding a transformation operation to be applied to all data in a dataset
- *Pipeline composition:* operation of declaratively constructing a composite transformation as a cascade of simpler transformations
- *Transformation application:* operation of actually constructing a new dataset from a prior dataset and a transformation

Predictor selection and definition. The next phase involves the *definition* of one or more predictors via a unique meta-activity, namely:

- *Predictor declaration:* operation of constructing a representation for a particular predictor, which implies choosing the predictor family and specifying actual values for its hyper parameters

Training. 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:

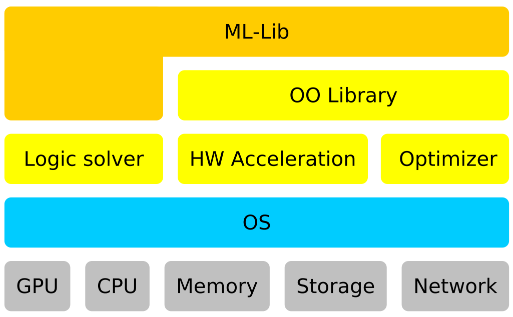

Figure 1: Layered view of the proposed ML-Lib. An OO library is assumed behind the scenes, providing high-level abstraction to optimize ML predictors, possibly via HW acceleration.

- *Predictor fitting w.r.t. a training set of data:* operation of fitting a predictors' internal parameter on some provided training data

Inference. Once in their *inference* phase, trained predictors may eventually be exploited to draw predictions. The corresponding activity is:

- *Predictor querying:* operation where (possibly unseen) values are provided to some trained predictor as a query, and the resulting values are interpreted as predictions

Validation. 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:* operation of computing a scoring value out of a trained predictor, a test dataset, and a scoring function

4. ML-Lib Overview and Architecture

This section discusses the design of ML-Lib, the logic programming library reifying the logic API for ML reifying the meta-model discussed above. The overall architecture is depicted in fig. 1. 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 for training and using ML predictors—other than any other entity involved in the process.

In the backend, the library assumes an underlying object-oriented (OO) library providing high-level ML abstractions, such as datasets, predictors, and so on. Examples of these libraries may be for instance Keras [12] or DeepLearning4J [9]. The OO library may in turn be backed by an optimizer taking care of making training and data management effective on the available hardware—and possibly exploiting hardware acceleration. 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

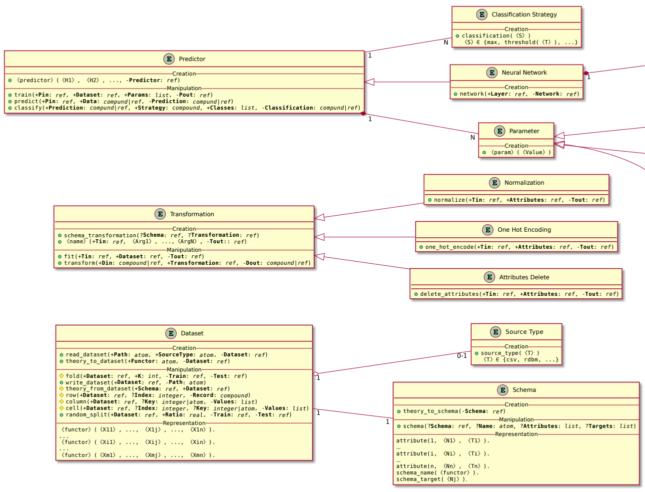
Predictor
Creation
⟨predictor⟩ (⟨H1⟩ , ⟨H2⟩ , ..., -Predictor: ref)
Manipulation
train(+Pin: ref, +Dataset: ref, +Params: list, -Pout: ref)
predict(+Pin: ref, +Data: compund|ref, -Prediction: compund|ref)
classify(+Prediction: compund|ref, +Strategy: compound, +Classes: list, -Classification: compund|ref)
Classification Strategy
Creation
classification(⟨S⟩)
⟨S⟩ ∈ {max, threshold(⟨T⟩), ...}
Neural Network
Creation
network(+Layer: ref, -Network: ref)
Layer
Creation
input_layer(+Size: int, -Layer: ref)
Manipulation
dense_layer(+Lin: ref, +Size: int, +Activatin: atom, -Lout: ref)
output_layer(+Lin: ref, +Size: int, +Activatin: atom, -Lout: ref)
Activation
Creation
activation(⟨A⟩)
⟨A⟩ ∈ {identity, relu, sigmoid, tanh, ...}
Parameter
Creation
⟨param⟩ (⟨Value⟩)
Loss
Creation
loss(⟨L⟩)
⟨L⟩ ∈ {mse, mae, cross_entropy, ...}
Max Epochs
Creation
max_epochs(+N: int)
Batch Size
Creation
batch_size(+N: int)
Transformation
Creation
schema_transformation(?Schema: ref, ?Transformation: ref)
⟨name⟩ (+Tin: ref, ⟨Arg1⟩ , ..., ⟨ArgN⟩ , -Tout:: ref)
Manipulation
fit(+Tin: ref, +Dataset: ref, -Tout: ref)
transform(+Din: compound|ref, +Transformation: ref, -Dout: compound|ref)
Normalization
normalize(+Tin: ref, +Attributes: ref, -Tout: ref)
One Hot Encoding
one_hot_encode(+Tin: ref, +Attributes: ref, -Tout: ref)
Attributes Delete
delete_attributes(+Tin: ref, +Attributes: ref, -Tout: ref)
Dataset
Creation
read_dataset(+Path: atom, +SourceType: atom, -Dataset: ref)
theory_to_dataset(+Functor: atom, -Dataset: ref)
Manipulation
fold(+Dataset: ref, +K: int, -Train: ref, -Test: ref)
write_dataset(+Dataset: ref, -Path: atom)
theory_from_dataset(+Schema: ref, +Dataset: ref)
row(+Dataset: ref, ?Index: integer, -Record: compound)
column(+Dataset: ref, ?Key: integer|atom, -Values: list)
cell(+Dataset: ref, ?Index: integer, ?Key: integer|atom, -Values: list)
random_split(+Dataset: ref, +Ratio: real, -Train: ref, -Test: ref)
Representation
⟨functor⟩ (⟨X11⟩ , ..., ⟨X1j⟩ , ..., ⟨X1n⟩).
...
⟨functor⟩ (⟨Xi1⟩ , ..., ⟨Xij⟩ , ..., ⟨Xin⟩).
...
⟨functor⟩ (⟨Xm1⟩ , ..., ⟨Xmj⟩ , ..., ⟨Xmn⟩).
Source Type
Creation
source_type(⟨T⟩)
⟨T⟩ ∈ {csv, rdbm, ...}
Schema
Creation
theory_to_schema(-Schema: ref)
Manipulation
schema(?Schema: ref, ?Name: atom, ?Attributes: list, ?Targets: list)
Representation
attribute(1, ⟨N1⟩ , ⟨T1⟩).
…
attribute(i, ⟨Ni⟩ , ⟨Ti⟩).
…
attribute(n, ⟨Nn⟩ , ⟨Tn⟩).
schema_name(⟨functor⟩).
schema_target(⟨Nj⟩).
1
N
0-1

Figure 2: Overview of our ML-Lib 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.

imply DeepLearning4J to be exploited, while targetting Python may exploit 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 the ML-Lib is provided in terms of logic predicates acting on the above defined entities. Details about the predicates are provided in the supplementary material. Figure 2 provides an overview of these predicates, grouped by entities.

5. ML-Lib Examples

Here we discuss the usage of the ML-Lib to serve the purposes described in section 2.

From an 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. Examples consist of Prolog scripts, possibly involving standard Prolog predicates.

From an ML perspective, our examples assume a very simple scenario where a neural-network classifier is trained on the well known Iris dataset [13]. The resulting NN is then exploited to write a simple hybrid predicate aimed at classifying unseen Iris instances.

Declarative ML. Declarativeness is a key benefit of our symbolic approach to ML. The ML-Lib supports declarative ML in several ways, as exemplified by listings 1, 2, 3, and 5.

In particular, listing 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

```prolog
% 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 1: Dataset loading from file

```prolog
%  declaring & fitting the preprocessing pipeline
preprocessing_pipeline(Dataset, Schema, Pipeline) :-
    schema_transformation(Schema, Step0),
    normalize(Step0, [petal_width, petal_length, sepal_width, sepal_length], Step1),
    one_hot_encode(Step1, [species], Step2),
    fit(Step2, Dataset, Pipeline).
```

Listing 2: Pre-processing pipeline

```prolog
% 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).
```

Listing 3: Neural network structure declaration

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 appendix A.1.1. 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 4 (presented later in this section) reports a similar scenario where the dataset as well is loaded from the local theory.

Listing 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 3 presents a general purpose predicate aimed at defining multi-layered perceptron predictors with an arbitrary amount of hidden layers. This is enabled by the `multi_layer_perceptron/4` predicate, which requires the caller to provide the number 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 number of neurons for each hidden layer – from the outermost to the innermost –, while the total amount of integers corresponds to the number of hidden layers. The resulting neural network predictor is then bound to the `NN` output argument. So, for instance, a NN having 4 input neurons, 2 hidden layers with 5 and 7 neurons respectively, and 3 output neurons can be declared as follows:

```
multi_layer_perceptron(4,[5, 7],3,NN)
```

Finally, listing 5 declares an end-to-end ML workflow aimed at selecting and training the best NN architecture to tackle Iris classification. It is worth noting that the declarative nature of the script can be regarded as a formal – yet human-readable – specification of a classifier training workflow.

Symbolic data sources. As highlighted above it may be useful to perform ML upon data expressed in logic form. This requires logic theories to act as symbolic data sources. ML-Lib supports such scenario, as exemplified in listing 4. The script is assumed to replace listing 4 in those situations where the Iris dataset is logically described in the 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 (listing 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

```prolog
/* attributes definition here */

% 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).
```

Listing 4: Dataset loading from the local theory

reference to the `Predictor` itself, but also its `Performance`, and the affine `Transformation` to be applied to each datum for which predictions should be drawn using that predictor. 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` – obtained by applying `Transformation` to the `TestSet` –, 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. Further details about the many predicates exploited in this example are provided in the supplementary material.

```
 1  /* Hyper paramenters */
 2  hidden_layers([10]). hidden_layers([20, 10]).
 3  hidden_layers([30, 20, 10]).
 4
 5  /* Learning paramenters */
 6  max_epochs(30). max_epochs(50).
 7  batch_size(32). batch_size(16).
 8  learning_rate(0.01). learning_rate(0.1).
 9  loss(cross_entropy).
10
11  /* Workflow paramenters */
12  target_performance(0.90). test_percentage(0.2).
13
14  /* Generates all hyper & learning params combinations */
15  params(
16    [hidden_layers(H)],
17    [iterations(X), learning_rate(Y), batch_size(Z), loss(L)]
18  ) :- hidden_layers(H), max_epochs(X), learning_rate(Y),
19       batch_size(Z), loss(L).
20
21  /* Generates and trains all Predictors for the given Dataset and Schema,
22  whose Performance is at least target_performance. */
23  model_selection(Dataset, Schema, Predictor, Transform, Performance) :-
24       test_percentage(R), target_performance(T),
25       random_split(Dataset, R, TrainSet, TestSet),
26       preprocessing_pipeline(TrainSet, Schema, Transform),
27       transform(TrainSet, Transform, ProcessedTrainSet),
28       params(HyperParams, LearnParams),
29       train_cv(ProcessedTrainSet, HyperParams, LearnParams, P),
30       P >= T,
31       multi_layer_perceptron(4, HyperParams, NN),
32       train(NN, TrainingSet, LearnParams, Predictor),
33       transform(TrainSet, Transform, ProcessedTestSet),
34       test(NN, ProcessedTestSet, Performance).
35
36  /* Example of training query: */
37  ?- iris_dataset(D), iris_schema(S), model_selection(D, S, P, _, A).
```

Listing 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 the supplementary material.

Under these hypotheses, a model selection workflow for the Iris dataset may be triggered via a concise logic query such as the one from listing 5 (line 37). If all aspects of the model selection workflow are correctly declared, the query provide multiple successful solutions corresponding to all trained predictors (P) and their test-set accuracies (A).

Hybrid reasoning. Finally, listing 6 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

```
 1  /* assumption: */
 2  :- iris_dataset(D), iris_schema(S), model_selection(D, S, N, T, _), !,
 3      assert(iris_nn(N, T)).
 4  
 5  /* hybrid iris classifier */
 6  iris(SL, SW, PL, PW, Species) :-
 7     iris_nn(Network, Transformation),
 8     transform([SL, SW, PL, PW] , Transformation, ActualX),
 9     predict(Network, ActualX, Y),
10     classify(Y, argmax, [setosa, versicolor, virginica], Species).
```

Listing 6: Exploitation of the NN classifier trained in listing 5 to create an hybrid predicate

```
iris(SL, SW, PL, PW, setosa) :- PW =< 0.78.
iris(SL, SW, PL, PW, versicolor) :- PL >= 2.86, PL < 4.91.
iris(SL, SW, PL, PW, virginica).
```

Listing 7: A purely symbolic classifier for Iris flowers, functionally equivalent to the hybrid one from listing 6

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 one from listing 6 (line 2) 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 6. 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 indistinguishable from a purely symbolic predicate serving the same purpose (i.e., Iris classification) and having the same name and arity—such as the one described in listing 7.

6. Conclusions

In this paper, 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 that any goal-oriented solver may support, there including Prolog ones. Both the syntax and the semantics of each predicate are discussed, as well as architectural and technological requirements. 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 that 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 of 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.

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 into 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.

Acknowledgments

This paper has been partially supported by *(i)* the European Union's Horizon 2020 research and innovation programme under G.A. no. 101017142 (StairwAI project), and by *(ii)* the CHIST-ERA IV project CHIST-ERA-19-XAI-005, co-funded by the EU and the Italian MUR (Ministry for University and Research).

References

[1] E. Ilkou, M. Koutraki, Symbolic vs sub-symbolic ai methods: Friends or enemies?, in: CIKM (Workshops), 2020.

[2] R. Calegari, G. Ciatto, A. Omicini, On the integration of symbolic and sub-symbolic techniques for XAI: A survey, Intelligenza Artificiale 14 (2020) 7–32. doi:10.3233/IA-190036.

[3] A. Barredo Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, F. Herrera, Explainable explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI, Information Fusion 58 (2020) 82–115. doi:10.1016/j.inffus.2019.12.012. arXiv:1910.10045.

[4] B. Goertzel, Perception processing for general intelligence: Bridging the symbolic/subsymbolic gap, in: J. Bach, B. Goertzel, M. Iklé (Eds.), Artificial General Intelligence, Springer Berlin Heidelberg, 2012, pp. 79–88.

[5] R. Calegari, G. Ciatto, S. Mariani, E. Denti, A. Omicini, LPaaS as micro-intelligence: Enhancing IoT with symbolic reasoning, Big Data and Cognitive Computing 2 (2018). doi:10.3390/bdcc2030023.

[6] R. Calegari, G. Ciatto, J. Dellaluce, A. Omicini, Interpretable narrative explanation for ML predictors with LP: A case study for XAI, in: F. Bergenti, S. Monica (Eds.), WOA 2019 – 20th Workshop "From Objects to Agents", volume 2404 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, RWTH Aachen University, 2019, pp. 105–112. URL: http://ceur-ws.org/Vol-2404/paper16.pdf.

[7] G. Ciatto, R. Calegari, A. Omicini, 2P-Kt: A logic-based ecosystem for symbolic AI, SoftwareX 16 (2021) 100817:1–7. doi:10.1016/j.softx.2021.100817.

[8] P. Körner, M. Beuschel, J. Barbosa, V. S. Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, G. Ciatto, Fifty years of Prolog and beyond, Theory and Practice of Logic Programming (2022) 1–83. doi:10.1017/S1471068422000102.

[9] Konduit, Deeplearning4J: Open-source distributed deep learning for the JVM, https://deeplearning4j.konduit.ai, 2022. (Supported by the Eclipse Foundation).

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: http://tensorflow.org/.

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

[12] F. Chollet, et al., Keras, https://keras.io, 2015.

[13] R. A. Fisher, Iris data set, https://archive.ics.uci.edu/ml/datasets/iris, 1936. (From the UCI Machine Learning Repository).

A. Supplementary Material

A.1. Realising the API: ML-Lib Design

In the reminder of this section, we adopt the following notation to denote the interfaces of logic predicates:

$$\texttt{functor(}\odot_1\ \texttt{Name}_1\texttt{:}\ \textit{type}_1\texttt{, . . . ,}\ \odot_N\ \texttt{Name}_N\texttt{:}\ \textit{type}_N\texttt{)}$$

where N denotes the arity of predicate `functor/`N, whose i^{th} argument – named $\texttt{Name}_i$ – must be of type $\textit{type}_i$, 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 union types ($\texttt{T}_1\,|\,\texttt{T}_2\,|\,\ldots$). 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 3.2, 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.

A.1.1. 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:

$$\begin{array}{l}
\texttt{attribute(1, } N_1\texttt{, } T_1\texttt{).}\\
\quad\vdots\\
\texttt{attribute(}i\texttt{, } N_i\texttt{, } T_i\texttt{).}\\
\quad\vdots\\
\texttt{attribute(}n\texttt{, } N_n\texttt{, } T_n\texttt{).}\\
\texttt{schema\_name(}N\texttt{).}\\
\texttt{schema\_targets([}N_j\texttt{, } N_k\texttt{, } \ldots\texttt{, } N_h\texttt{]).}
\end{array}$$

[1] cf.https://www.swi-prolog.org/pldoc/man?section=preddesc

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`.

A.1.2. 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:

$$\begin{gathered} N(\mathtt{X}_{1,1},\ \ldots,\ \mathtt{X}_{1,j},\ \ldots,\ \mathtt{X}_{1,n}). \\ \vdots \\ N(\mathtt{X}_{i,1},\ \ldots,\ \mathtt{X}_{i,j},\ \ldots,\ \mathtt{X}_{i,n}). \\ \vdots \\ N(\mathtt{X}_{m,1},\ \ldots,\ \mathtt{X}_{m,j},\ \ldots,\ \mathtt{X}_{m,n}). \end{gathered}$$

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 $\mathtt{K} \geq 2$, 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 are all 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.

A.1.3. 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:

$\langle name \rangle$`(+Pipeline`$_{in}$`: ref, +`A_1`, . . . , +`A_n`, -Pipeline`$_{out}$`: ref)`

where $\langle name \rangle$ is the name of the transformation being appended to `Pipeline`$_{in}$, while $A_1, \ldots, A_n$ are transformation-specific parameters, and `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,T_0),
transformation_1(T_0, arg_1, T_1),
    ⋮
transformation_m(T_{m-1}, arg_m, T_m),
schema_transformation(FinalSchema, T_m)
```

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 $\mathtt{T}_i$, 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:

$$\texttt{transform(?Data}_{in}\texttt{: } \mathit{ref}\,|\,\mathit{compound}\texttt{, +Transformation: } \mathit{ref}\texttt{, ?Data}_{out}\texttt{: } \mathit{ref}\,|\,\mathit{compound}\texttt{)}$$

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`$_{in}$.

A.1.4. 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:

$$\langle \mathit{name} \rangle\texttt{(+}H_1\texttt{, . . . , +}H_n\texttt{, -Predictor: } \mathit{ref}\texttt{)}$$

where $\langle \mathit{name} \rangle$ 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:

`train(+Predictor`$_{in}$`: ` *`ref`*`, +Dataset: ` *`ref`*`, +Params: ` *`list`*`, -Predictor`$_{out}$`: ` *`ref`*`)`

The predicate accepts `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 `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 `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:

$$\langle name \rangle (\langle value \rangle)$$

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;
- `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`*`)`

[2] i.e., the amount of times the learning algorithm works through the entire training dataset

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 $\mathbf{x}$ 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:

$\langle name \rangle$`(+Actual: `*`ref|list`*`, +Expected: `*`ref|list`*`, -Score: `*`real`*`)`

where $\langle name \rangle$ 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.

A.1.5. 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`*`).`
- $\langle type \rangle$`_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, $\langle type \rangle$`_layer/4` is matched by a number of actual predicates aimed at creating intermediate or output layers. There $\langle type \rangle$ 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 $\langle type \rangle$`_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.

B. Model selection: further details

The model selection example discussed in section 5 and formally described in listing 5 relies upon a number of ancillary predicates declaring some particular steps of the workflow and exemplifying many ML-Lib functionalities. These are reported in listing 8. 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. Every single fold of a `K`-fold CV process is managed

```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).
```

Listing 8: Ancillary predicates used in listing 5. Each predicate denotes one particular step of a model selection workflow

by the `train_cv_fold/5` predicate, which in turn exploits `train_validate/5` predicate to train and validate every 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 an argument.