Probabilistic Rule Learning Systems: A Survey

ABDUS SALAM, ROLF SCHWITTER, and MEHMET A. ORGUN, Macquarie University

This survey provides an overview of rule learning systems that can learn the structure of probabilistic rules for uncertain domains. These systems are very useful in such domains because they can be trained with a small amount of positive and negative examples, use declarative representations of background knowledge, and combine efficient high-level reasoning with the probability theory. The output of these systems are probabilistic rules that are easy to understand by humans, since the conditions for consequences lead to predictions that become transparent and interpretable. This survey focuses on representational approaches and system architectures, and suggests future research directions.

CCS Concepts: • **Computing methodologies → Rule learning**; **Logical and relational learning**; **Probabilistic reasoning**;

Additional Key Words and Phrases: Probabilistic rule learning, symbolic rule learning, sub-symbolic rule learning, probabilistic logic programming

ACM Reference format:
Abdus Salam, Rolf Schwitter, and Mehmet A. Orgun. 2021. Probabilistic Rule Learning Systems: A Survey. *ACM Comput. Surv.* 54, 4, Article 79 (April 2021), 16 pages.
https://doi.org/10.1145/3447581

1 INTRODUCTION

Recently, symbolic approaches to machine learning gained interest because of their comprehensibility and interpretability [24]. These approaches are also known as **Inductive Logic Programming (ILP)** and can be used to learn rules from observed examples and background knowledge [18, 22]. The learned rules can then be used to make predictions for unseen examples. The observed examples represent the relevant facts of the problem at hand and provide evidence for the learning process. The contextual information of the problem domain is represented as background knowledge. However, in ILP, all of this information is deterministic; that means we cannot represent uncertain information.

Most ILP systems such as FOIL [26], Progol [19], and Metagol [4] learn deterministic rules, whereas only a few of these systems, such as Aleph [40], provide confidence values for the learned rules. **Probabilistic Logic Programming (PLP)** introduces probabilistic reasoning into logic programs to represent uncertain information [7, 31]. **Probabilistic Inductive Logic Programming (PILP)**, which is also sometimes called *Statistical Relational Learning* [29], combines PLP and ILP

Authors' address: A. Salam, R. Schwitter, and M. A. Orgun, Department of Computing, Macquarie University, Balaclava Road, North Ryde, Sydney, New South Wales, 2109, Australia; emails: {abdus.salam, rolf.schwitter, mehmet.orgun}@mq.edu.au.

0360-0300/2021/04-ART79 $15.00
https://doi.org/10.1145/3447581

to learn probabilistic rules [6, 33]. It can take probabilistic information (examples and background knowledge) as input and induces rules with probabilities as output. Two types of learning tasks can be performed in PILP: parameter learning and structure learning. In parameter learning, the probability of an existing rule is learned, whereas in structure learning, the entire structure of the rule together with its probability is learned. In this survey, we focus on structure learning, as it subsumes parameter learning.

Several PILP systems exist, such as PRISM [38], ProbFOIL [5], and *cplint* [32], that use different learning approaches for learning rules inductively. In contrast to these symbolic PILP systems, there are also several sub-symbolic systems, such as **Neural Logic Programming (Neural LP)** [45] and Neural Theorem Provers [35], that use a neural network approach to learn rules from examples and background knowledge. Another group of systems, such as DeepProbLog [15], LIME-Aleph [27], and NLProlog [43], are hybrid systems, as they combine symbolic and sub-symbolic approaches to learn rules using a similar setting as the ILP systems. Although the sub-symbolic and the hybrid systems are not considered as ILP systems, they can learn the same type of logical rules using the same type of examples and background knowledge as ILP systems do. This is why we have included these sub-symbolic and hybrid rule learning systems in our survey as well. Despite their architectural differences, all of these systems use examples and background knowledge for learning.

There are surveys related to the PLP systems that focus on the implementation of probabilistic programming concepts and on the calculation of probabilities employing the distribution semantics and logical inference [7, 34]. Other surveys review hybrid systems and their features and look at techniques that combine symbolic and sub-symbolic representations [11, 28]. In contrast to these surveys, our study discusses probabilistic rule learning systems with respect to their input format and architecture with a particular focus on rule learning. We discuss probabilistic rule learning systems from two different perspectives: (1) the way in which they represent the input information (examples and background knowledge) and (2) the way in which they construct rules algorithmically with respect to their architectures. One of the aims of this survey is to assist readers in selecting the most suitable system depending on their requirements.

2 EXAMPLE SCENARIO

In the following, we use a simple scenario from the weather domain to illustrate the features of different rule learning systems. The scenario is adopted from one of the Aleph examples available in GitHub.[1] In our scenario, a binary prediction is made whether a game is played or not on a particular day based on the weather condition; the prediction will be either `play` or `not_play` for that day. The prediction is made based on the following weather information (attributes):

- *Outlook*: This attribute provides the information about the outlook of the day. The value of this attribute can be `sunny`, `overcast`, or `rain`.
- *Windy*: This attribute states whether it is windy or not on a particular day. The value of this attribute can be `true` or `false`.
- *Temp*: This attribute represents the temperature of a day. The value of this attribute is an integer.
- *Humidity*: This attribute represents the humidity of a day. The value of this attribute is a positive integer.

[1] https://github.com/friguzzi/aleph.

Table 1. List of Predicates Used for the Weather Example

Predicate	Description
`class/2`	Represents the prediction
`outlook/2`	Represents the *outlook* attribute
`windy/2`	Represents the *windy* attribute
`temp/2`	Represents the *temp* attribute
`humidity/2`	Represents the *humidity* attribute

A predicate is expressed as `predicate/arity`, where `predicate` and `arity` represent the name and the number of arguments of the predicate, respectively.

Here is an example instance of the data for a particular day: `day: d1`, `outlook: sunny`, `temp: 75`, `humidity: 70`, `windy: true`, `prediction: play`. This example tells us that on day `d1`, the prediction is `play` for the given attribute values.

We use one predicate to represent the prediction and four predicates to specify the attributes and their values (Table 1). All of these predicates have two arguments: the first argument specifies a day, and the second argument specifies the attribute value or the prediction value. If we consider the preceding example instance, then we obtain the following predicates: `outlook(d1, sunny)`, `temp(d1, 75)`, `humidity(d1, 70)`, `windy(d1, true)`, and `class(d1, play)`.

Using this example scenario, in the following sections we discuss the settings that we use for learning rules in the probabilistic rule learning systems. We consider rules of the following form:

$$\texttt{h :- } \texttt{b}_1\texttt{, b}_2\texttt{, ..., b}_n\texttt{,}$$

where the logical literal `h` is an atom[2] that forms the head of the rule and the conjunction of the logical literals $\texttt{b}_i$ form the body of the rule. If the body is empty (`n=0`), then the rule is considered as a fact. The following rule represents the preceding example instance:

```
class(d1, play) :-
  outlook(d1, sunny), temp(d1, 75),
  humidity(d1, 70), windy(d1, true).
```

In this rule, all arguments of all literals are constants. However, usually in rule learning, we try to learn rules that are as general as possible. If we want to express a more general rule that we can apply to any instance of a day, then we can use variables instead of constants.

PLP uses probabilities to represent the uncertainty about the information of a problem. Different PLP systems structure statements in a different way to represent the probability associated with a fact or a rule. For example, *cplint* [32] uses a colon (`:`) to attach a probability (`P`) to a fact (`fact:P`) or to a rule (`head:P :- body`). Based on this, the fact `outlook(d1, sunny):0.4` specifies that day `d1` is `sunny` with the probability of 40%. Similarly, the preceding rule can be annotated with a probability—for example: `class(d1, play):0.6 :-body`, where the prediction for day `d1` is `play` with the probability of 60%.

In our dataset, we have data for 14 days. For 9 days, the prediction is `play` and the prediction is `not_play` for the rest of the days. As mentioned earlier, we need four predicates to specify the weather information of a day and another predicate to state the prediction for the day (see Table 1). Hence, to make a prediction for a day, we would like to learn rules as discussed in this section where the predicate `class/2` appears in the head and all other predicates appear in the

[2]Atoms are predicate symbols together with their arguments, each argument being a term.

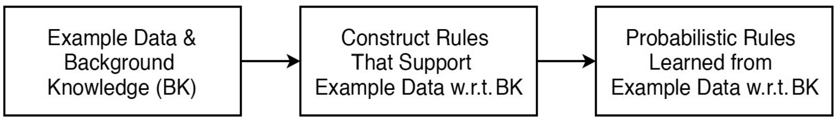
Example Data & Background Knowledge (BK)
Construct Rules That Support Example Data w.r.t. BK
Probabilistic Rules Learned from Example Data w.r.t. BK

Fig. 1. Generic rule learning process.

body of the rules. We use the structure of this simple rule as a starting point for the following discussion of the rule learning process.

3 GENERIC RULE LEARNING PROCESS

Rule learning systems use different approaches to construct rules. However, when we have a closer look at these systems and compare their algorithms, we can observe a common process for learning rules (Figure 1).

As a starting point, all systems require background knowledge and a set of examples. The background knowledge represents the deterministic information. An example is an observation that represents data related to an instance. The example can be positive or negative. If a prediction for an instance holds in a scenario, then this information is represented as a positive example; otherwise, if the prediction does not hold, it is represented as a negative example. For example, in our weather scenario, if an observation represents data for the prediction `play`, then this observation is considered to be a positive example for the prediction `play` and a negative example for the prediction `not_play`. Background knowledge and example instances account for the input information to the systems.

Next, the systems take this input information to construct the rules. Rules are constructed in such a way that they support as many positive examples as possible and no negative examples with respect to the background knowledge. Once a rule is constructed, the probability of this rule is calculated based on the number of positive and negative examples supported by the rule. Since the learned rules are probabilistic, they are annotated with probability values that correspond to the prediction values.

Some ILP systems such as Aleph and *cplint* also require configurational settings that are used to guide the rule construction process. We also discuss these setting where applicable.

4 PROBABILISTIC RULE LEARNING SYSTEMS

In this section, we discuss four probabilistic rule learning systems. These systems have been selected because of their representational approaches. The systems Aleph and *cplint* use a symbolic representation, whereas Neural LP uses a sub-symbolic representation, and LIME-Aleph uses a combination of these representations.

4.1 Aleph

Aleph stands for "A Learning Engine for Proposing Hypotheses" and is an ILP system written in Prolog that is capable of learning rules from positive and negative examples [40]. Aleph is available for both Yap-Prolog[3] and SWI-Prolog.[4] In general, Aleph requires as input three data files with different file extensions that represent the background knowledge, positive examples, and negative examples. To learn rules, Aleph reads all of the data from these files and then constructs rules using the data. In the SWI-Prolog version of Aleph, we can use a single program file instead

[3]https://github.com/vscosta/yap-6.3.
[4]https://www.swi-prolog.org/.

of three separate files for rule learning. In the following, we describe the learning process of Aleph based on the SWI-Prolog version.

As we have seen previously, an example instance consists of a number of facts that are used to build the head and the body of a rule. In Aleph, only facts that are related to the head of a rule are used as examples to construct the head literal, whereas the facts in the background knowledge are used to construct the body literals.

To construct a rule, Aleph starts with the most specific rule built from a positive example and then tries to find a more general rule. In the rule learning process, Aleph selects a positive example from the list of given positive examples, uses the information defined in the program settings, and constructs the most specific rule that entails the selected example. Afterward, Aleph tries to find a more general rule using a subset of the body literals from the most specific rule. While searching for this more general rule, Aleph tries to find a set of candidate rules. For each candidate, a score is calculated based on the number of positive and negative examples that are supported. Next, the candidate with the highest score is selected as the general rule. The positive example that has been used to generate the general rule is removed from the list of positive examples, and this process is repeated until the list is empty.

Alternatively, Aleph can learn rules only from positive examples. In this case, a rule is evaluated using the Bayesian score [20] that is calculated based on Bayes' theorem where the probability of a rule is its posterior estimation given the number of positive examples. For our weather example, we have used both positive and negative examples to learn the rules. The positive examples are written in the form of facts between two directives: `:- begin_in_pos` and `:- end_in_pos` and the negative examples between: `:- begin_in_neg` and `:- end_in_neg`. In our case, we only have positive and negative examples of the predicate `class/2` (Listing 1), because this predicate ends up in the head of a rule.

Listing 1. Sample positive examples in Aleph.

```
:- begin_in_pos.
   class(d1, play).
   class(d2, not_play).
:- end_in_pos.
```

The domain-specific information of the problem and the facts related to the body literals of the rules are specified in the background knowledge section of an Aleph program. The background knowledge is written in the form of clauses between two directives `:- begin_bg` and `:- end_bg`. In our example, the attributes (see Table 1) of the weather information are provided as facts in the background knowledge section (Listing 2), as the predicates representing these attributes can appear in the body of a rule.

Listing 2. Sample background knowledge in Aleph.

```
:- begin_bg.
   outlook(d1, sunny).
   temp(d1, 75).
   humidity(d1, 70).
   windy(d1, true).
:- end_bg.
```

As mentioned earlier, Aleph requires program settings for learning; these settings are known as language restrictions. These language restrictions are expressed via mode declarations and determination statements. Mode declarations (`mode/2`) are used to define those predicates that can occur in a rule to be learned. The first argument of `mode/2` specifies the recall value and the second argument specifies the template of those predicates that can participate in a rule. The recall value can be a positive integer or an asterisk (`'*'`). An integer indicates how many facts in the examples are called for learning a predicate of a rule, whereas the value `'*'` stands for an unbounded number of calls. A predicate template has the following format: pred(arg_1, arg_2, ..., arg_n), where pred is the predicate name and arg_i is of one of the following three types: an input variable, an output variable, or a constant. The symbols "+," "–," and "#" in the prefix of an argument specify that it is either an input or an output variable or a constant. In our context, we have to use mode declarations for all of those predicates that can occur in a rule to be learned (Listing 3).

After declaring the mode of these predicates, determination statements (determination/2) need to be specified. The first argument of these determination statements describes which predicate can occur in the head of a rule (e.g., class/2), and the second argument describes which predicates (e.g., outlook/2) can occur in the body (see Listing 3). Only one target predicate can be defined in the determination statements at a given point in time. That means Aleph can only learn rules that contain the same head, but these rules can contain different bodies.

Listing 3. Sample mode declarations and determination statements in Aleph.

```
:- mode(1, class(+day, -class)).
:- mode(1, outlook(+day, #outlook)).
:- determination(class/2, outlook/2).
:- determination(class/2, temp/2).
```

To define a parameter related to the learning algorithm, we can use the predicate aleph_set/2: the first argument of this predicate specifies the parameter name, and the second one specifies its value. One important parameter name that we have to specify in our context is classes (Listing 4). This parameter name declares the possible prediction values play and not_play, as Aleph needs to learn the probabilities for all predictions together with the rules.

Listing 4. Sample parameters in Aleph.

```
:- aleph_set(classes, [play, not_play]).
:- aleph_set(tree_type, class_probability).
```

After specifying the example instances, the background knowledge, and the language restrictions, the Aleph program is ready for rule learning. The three most important commands are as follows:

- The command induce/1 uses the basic Aleph algorithm, learns the rule, and returns the learned rule as output.
- The command induce_incremental/1 allows Aleph to learn a rule in an interactive way, taking input from a domain expert.
- The command induce_tree/1 uses a tree-based construction algorithm to learn a rule.

The command induce_tree/1 also provides a number of options—two of which, regression and class_probability, support learning probabilistic rules. If one of these options is selected, then the learned rules contain probability values for different predictions that are expressed by an additional predicate (random/2) in the body of the rule. For our weather example, we have used the command induce_tree/1 together with the class_probability option because our goal is to learn probabilistic rules. The rules learned by Aleph are shown in Listing 5, where the predicate random/2 in the first rule specifies that the probabilities are 0.25 and 0.75 for the predictions play and not_play (mentioned in the variable B), respectively, when all information about a particular day (mentioned in the variable A) satisfies the rule.

Listing 5. Sample rules learned in Aleph.

```
class(A, B):-
  outlook(A, rain),
  windy(A, true),
  random(B, [0.25-play, 0.75-not_play]).

class(A, B):-
  not((outlook(A, rain), windy(A, true))),
  random(B, [0.72-play, 0.28-not_play]).
```

4.2 *cplint*

Another PLP system is *cplint*, which facilitates inference and learning of probabilistic logic programs from examples and background knowledge [32]. As Aleph, *cplint* is also available in YAP-Prolog and SWI-Prolog. *cplint* employs the syntax of **Logic Programs with Annotated Disjunctions (LPADs)** [41], which is a general syntax for PLP under the distribution semantics [37]. *cplint* has a number of libraries for parameter and structure learning. In the following, we describe the *cplint* rule learning process based on the SLIPCOVER algorithm. SLIPCOVER is an algorithm for learning both the probabilities and the structure of probabilistic rules expressed as an LPAD program by performing a beam search in the space of clauses and a greedy search in the space of theories [2]. There are different parts of a *cplint* program that we need to specify according to the characteristics of the learning problem.

In a *cplint* program, the observed data is defined with the help of a number of example model instances. Such an instance starts with `begin(model(<name>))` and ends with `end(model(<name>))`, and contains the observed facts between these directives. In contrast to Aleph, all facts that are related to the head and body literals of a rule are specified in the example model instances. In addition, we can pre-define the probability for an example instance using the predicate `prob(P)`. In this case, the probability `P` is used in the learning process. If an example instance does not have a pre-defined probability, then the probability $\frac{1}{n}$ is assigned to this instance, where `n` is the total number of example instances. The predicate `neg/1` is used to define the negation of a fact in an example instance. In our weather example, we use 14 example instances where each instance consists of facts that are expressed using the predicates in Table 1 (Listing 6).

Listing 6. Sample example instance in *cplint*.

```
begin(model(w1)).
  class(d1, play).
  neg(class(d1, not_play)).
  outlook(d1, sunny).
  temp(d1, 75).
  humidity(d1, 70).
  windy(d1, true).
end(model(w1)).
```

The problem specific deterministic clauses (clauses that are true for all example instances) are defined in the background knowledge of a *cplint* program. The *cplint* program represents the background knowledge between the two directives `:- begin_bg` and `:- end_bg`. For our weather example, the background knowledge section is empty, as we do not have such information.

Similar to the language restrictions of Aleph, *cplint* uses a language bias for mode declarations and determination statements, but in addition it requires input-output declarations. Instead of using a single predicate (`mode/2`) for mode declarations like in Aleph, *cplint* distinguishes two predicates: `modeh/2` and `modeb/2`. The predicate `modeh/2` specifies which predicate can occur in the head of a rule and the predicate `modeb/2` specifies which predicates can occur in the body.

Both predicates have the same structure, which is similar to the structure of the predicate `mode/2` in Aleph. In our weather example, the target predicate `class/2` is declared using `modeh/2` and the rest of the predicates are declared using `modeb/2` (Listing 7).

Listing 7. Sample mode declarations and determination statements in *cplint*.

```
modeh(1, class(+day, -#class)).
modeb(1, outlook(+day, -#outlook)).
determination(class/2, outlook/2).
determination(class/2, temp/2).
```

The determination statements of a *cplint* program follow the same structure as in Aleph. However, *cplint* can have multiple target predicates in the determination statements, as it can learn rules for multiple target predicates.

For output declarations, an output predicate is used to declare the target predicates for which the learning algorithm will be applied; it has the form: `output(<predicate>/<arity>)`. Similarly, for input declarations, input predicates that can occur in the body of a rule must be declared. These input predicates have either the form `input(<predicate>/<arity>)` or `input_cw(<predicate>/<arity>)`. The predicate `input/1` declares an open world predicate, whereas the predicate `input_cw/1` declares a closed world predicate. Under the open world assumption [17], the predicate given as the argument of `input/1` participates in the learning process together with those predicates that can be derived from that argument. Under the closed world assumption [17], only the predicate given as the argument of `input_cw/1` participates in the learning process. In our weather example, we need to declare the target predicate `class/2` as an output predicate. All other predicates can occur in the body of a rule and are declared as input predicates using `input_cw/1`, as we want to use only the predicates specified in the example instances (Listing 8).

Listing 8. Sample input and output declarations in *cplint*.

```
output(class/2).
input_cw(outlook/2).
input_cw(temp/2).
```

The preamble of a *cplint* program is usually defined at the beginning of the program. In the preamble, we load all necessary libraries, initialize the algorithm, and specify the relevant parameters for the algorithm. For our weather example, we have to load and initialize the SLIPCOVER library and set all necessary parameters.

In *cplint*, the predicates `induce_par/2` and `induce/2` are used for parameter and structure learning. For our weather example, we execute the command `induce([train], R)` to learn the structure of the rules together with their probabilities. The first argument is a fold that consists of a list with all example instances. The learned rules are returned in the second argument `R`. Listing 9 shows the learned rules in *cplint* where the first rule specifies that the probability for the prediction `play` is `0.33`, if all information about a particular day (mentioned in the variable `A`) satisfies the rule.

Listing 9. Sample rules learned in *cplint*.

```
class(A, play):0.33 :-
  outlook(A, sunny),
  windy(A, false).

class(A, not_play):0.67 :-
  outlook(A, sunny),
  windy(A, false).
```

4.3 Neural LP

Neural LP is a differentiable system that supports structure learning to learn probabilistic rules based on a gradient-based programming framework [45]. It is implemented in Python and available via GitHub.[5]

Neural LP is an extension of a differentiable probabilistic logic called *TensorLog* [3] that can only perform parameter learning. Neural LP has been designed to learn rules for a particular knowledge-based reasoning task. In this context, a knowledge base is simply a collection of relations of the form: `relation(entity1, entity2)` that express binary relations. Here, an entity is an instance of an object. The learning task involves a query of the same format where the first argument is a variable and the second argument is a constant. Neural LP tries to find entities that satisfy the query. For our weather example, we use a query of the form `class(A, d1)` to find the prediction value (in variable `A`) for day `d1`. To do this, Neural LP first learns probabilistic rules for the query and then searches for those entities that comply with the first argument in the query while the entity in the second argument is fixed. The probabilistic rules assist to produce a ranked list of the entities associated with the corresponding scores where a higher score indicates that the entity is more relevant for answering the query. We can consider the knowledge base in Neural LP as the background knowledge that we have described in the generic rule learning process (see Section 3). The relations can be viewed as predicates from a logical point of view.

Neural LP requires three files to define the background knowledge (knowledge base):

- All entities that are used in the learning problem are declared in the file *entities*. Each line of this file contains a constant.
- All relation names of the problem domain are defined in the file *relations*.
- All known facts are specified in the file *facts*. Each line of this file contains a fact consisting of the following tab separated format: `entity1 predicate entity2`. All predicates that are used to construct the body of a rule are represented in this file.

For our weather example, attribute values (e.g., `sunny` and `rain`) and the prediction values (`play` and `not_play`) are declared in the file *entities*. The predicate names in Table 1 are defined in the file *relations*, and all facts except those related to the target predicate are specified in the file *facts* (Listing 10).

Listing 10. Sample facts in Neural LP.

```
d1    outlook  sunny
d1    windy    true
```

To specify the example instances, Neural LP uses three files: *train*, *test*, and *valid*. These files use the same format as the file *facts* to specify the example instances. Like the example instances in Aleph, only those facts that are used to construct the head of a rule are used as example instances. The rules are learned using the example instances specified in the file *train*. The example instances in the file *test* and the file *valid* are used to calculate the accuracy of the learned rules and for early termination of the learning process. In our weather example, all facts of the target predicate are specified in these files (Listing 11).

Listing 11. Sample example instances in Neural LP.

```
d1    class    play
d2    class    not_play
```

[5]https://github.com/fanyangxyz/Neural-LP.

After preparing these files with the required information, we execute the Neural LP program that learns the rules and stores them in a file. Neural LP can only learn chain rules for binary relations. A rule with the head `R(X,Y)` is an elementary chain rule if variables in its body form a chain from `X` to `Y`.

4.4 LIME-Aleph

The LIME-Aleph system [27] is different from the systems discussed so far in the sense that it does not directly learn the rules. The system is used to explain the prediction of an example instance, and to do so it learns rules and uses them for explanation. That is why we consider the LIME-Aleph system as a rule learning system. The implementation of this system is not publicly available.

LIME-Aleph uses two existing systems—LIME [30] and Aleph [40]—to generate the rules for explanation in a classification task. LIME (Local Interpretable Model-Agnostic Explanations) can explain the prediction for an example instance by selecting a number of attributes that are considered important for making the prediction [30].

LIME-Aleph assumes that we have example data in a tabular format that we can feed to the classifier (actually any form of classifier). Each column of the table represents an attribute, and each row represents data for a particular instance. After the classifier has been trained, it can be used to make predictions for the example instances. Next, the classification model, the example instance, and a number of k attributes are sent to LIME. LIME returns those attributes that are considered the most important ones for making the prediction. For the returned attributes, LIME-Aleph extracts all possible values and uses these values to find relations from a list of fixed relations that hold between the attributes. The important attributes and their values, as well as the relations and their values, are represented as predicates. Those predicates that have been derived from a record in the table form an example instance. The example instances found in this step are collected in a positive example list. For each relation found between important attributes, perturbed example instances are generated from the example instance by altering the attribute values and flipping the values of the relations. A perturbed example instance is sent to the model which returns an evaluation value. If the evaluation value is above a certain threshold for the prediction value, then the perturbed example is added to the positive example list; otherwise, the perturbed example is added to the negative example list. The new relations that are produced while generating the perturbed examples are added to the background knowledge. Finally, the positive and negative example lists are sent to Aleph for rule learning. The learned rules are then used to explain the example instance.

As discussed in this section, LIME-Aleph uses Aleph to learn deterministic rules, but the same architecture can also be used to learn probabilistic rules as discussed in Section 4.1.

5 SYSTEM ARCHITECTURES

From an architectural point of view, we can distinguish three types of rule learning systems: symbolic, sub-symbolic, and hybrid systems. In the following, we give an overview about the architectures of these different types of systems.

5.1 Symbolic Systems

Most of the symbolic probabilistic rule learning systems use discrete search in the problem space to find the expected rules. These systems start with a rule or a set of rules that support the examples with respect to the background knowledge. Afterward, they construct a candidate set of rules using different search techniques (e.g., beam search in SLIPCOVER [2]). The rules are constructed with the help of a language bias that specifies a template of the rule to be learned. For each candidate rule, an evaluation value is calculated to evaluate the rule. For example, SLIPCOVER uses log likelihood for the evaluation. Log likelihood is calculated from the positive and negative examples by using an expectation maximization algorithm known as EMBLEM [1]. From the set of candidate rules, only those rules that satisfy a certain threshold are selected. These selected rules form the final set of rules that build a theory. Afterward, parameter learning is performed for this final set of rules to learn their probabilities. Existing symbolic systems use different parameter learning algorithms [5, 10]. SLIPCOVER uses the same algorithm as EMBLEM for parameter learning.

5.2 Sub-symbolic Systems

Like a symbolic system, a sub-symbolic probabilistic rule learning system takes the same information as input but performs the task of structure learning in a different way. Sub-symbolic systems do not rely on grounding techniques used by symbolic systems for inferencing. Instead, they use a vector or matrix representation to specify the information for a predicate. For example, Neural LP uses a matrix to represent a binary predicate where each row and column index pair is associated with an entity. The value of a cell in the matrix is 1 if the predicate is true for the entities that correspond to the row and the column of the cell. After representing the information in the vector-matrix space, these systems usually try to find a list of candidate rules for each target predicate. The body of a candidate rule is constructed by combining the different predicates. For each candidate rule, these systems use different matrix operations to perform logical inference. The resulting matrix of these matrix operations forms a representation of the target predicate. The probability of a rule is calculated using the number of examples supported by the matrix that corresponds to the target predicate. For example, if we have a candidate rule that contains the atom h in the head and the literals b1 and b2 in the body, then Neural LP applies a matrix multiplication using b1 and b2 to find the values of the matrix for the atom h. If the result of the matrix multiplication supports the example instances, then that rule is selected and added to the final rule list.

5.3 Hybrid Systems

There is no standard way to combine symbolic and sub-symbolic representations. Hence, we observe that researchers have been developing hybrid systems employing different techniques to integrate these representations, and most of these systems are domain specific [9, 13–15, 27, 43]. We consider a system as a hybrid one if the system employs symbolic and sub-symbolic representations in separate components. In the following, we discuss three hybrid systems: LIME-Aleph [27], DeepProbLog [15], and NLProlog [43], which are based on different architectures with respect to representing the information in the components and communicating between the components. These systems use Horn clauses and vector representations in the symbolic and sub-symbolic components, respectively.

LIME-Aleph combines symbolic and sub-symbolic representations as the two main components and communicates via a controller. The sub-symbolic component performs a prediction task. To explain this prediction, the task of the symbolic component is to learn rules from positive and negative examples. The controller generates these examples from the original example instances by modifying influential attributes that have been selected algorithmically. To decide which modified example is a positive or negative one, the controller relies on the sub-symbolic component.

DeepProbLog learns a single task and integrates symbolic and sub-symbolic representations. It introduces neural predicates that apply sub-symbolic methods to make predictions with probabilities for prediction tasks. In DeepProbLog, a learning task is composed of several sub-tasks. Some of these sub-tasks may require to make predictions and are represented with the help of neural predicates. The symbolic component of the system represents a program for the learning task so that it can employ the required background knowledge. When the system evaluates the program, the probabilities of the general predicates are calculated from the given data and the probabilities of the neural predicates come from the sub-symbolic component of the system and are combined in the symbolic component.

NLProlog is a hybrid system specifically designed for natural language processing tasks that require multi-hop reasoning. The system converts a natural language statement into a triple consisting of two entities (e.g., *Socrates* and *Athens*) and a textual surface pattern (e.g., *ENT1 was born in ENT2*) that joins the entities in a statement. Such a triple is treated as a fact in the system. The *encoder* component of the system takes the facts, a query, and a number of rule templates as input and employs a sub-symbolic method to find the similarity scores between entities or textual surface patterns. The goal of the system is to learn a rule that answers the query. The system uses a *prover* component for this purpose that takes the similarity scores computed by the *encoder* as input, including the facts, the query, and the rule templates. The *prover* applies symbolic reasoning that uses backward-chaining with weak unification [39] to find the proofs for the query and computes unification scores based on the similarity scores. Each proof found by the *prover* is assigned a proof score from the aggregation of the unification scores. The proof with the highest score is selected as an answer to the query that constructs the learned rule.

6 DISCUSSION

The survey of the rule learning systems in Section 4 and their architectures in Section 5 lead us to a number of interesting observations and raise some related research questions.

Automate mode declarations. McCreath and Sharma [16] suggest an approach to automatically declare the mode for rule learning in symbolic systems. The mode information is extracted from the determination statements and the background knowledge. Aleph also has a similar feature for mode learning. However, the accuracy of the mode learning task is usually not good enough, and that is why the modes are still declared manually. Automating the mode declaration is a challenging task due to the difficulties of obtaining the contextual knowledge related to the problem that is not part of the problem description. But if it could be done with high accuracy, then writing a program to learn probabilistic rules in different PILP systems would require less human effort.

Simple rule templates for learning. Unlike the language bias of Aleph and *cplint*, Metagol utilizes meta-rules as templates for the rule learning process [4]. This approach requires a smaller amount of settings, but the resulting rules are not probabilistic. If an approach that is based on Meta-Interpretive Learning [23] can be used for structure learning in PILP systems, then the amount of work required for declaring the language bias can be reduced.

Improve learning time. Learning probabilistic rules in systems such as Aleph and *cplint* is usually computationally expensive. The time complexity increases exponentially with the size of the dataset because of the inference techniques that are used in these systems. To calculate the evaluation value for a candidate rule, these systems use grounding techniques [19] for inference. When these systems have to evaluate multiple rules, the same grounding may occur multiple times, which increases the time complexity. Lifted inference [25] is one of the techniques that has been proposed to overcome this problem. LIFTCOVER [10] and SafeLearner [12] are systems that use lifted inference to learn the probability of a rule. Let us assume we have a rule `h(X) :- b(X, Y)` and we want to calculate the probability for `h(obj)` where `obj` is a constant. To do this, lifted inference uses the total number of instances that satisfy `b(obj, Y)`. It can use this number to calculate the probability of a rule whenever `b(obj, Y)` occurs in the body of a rule. Hence, it does not need to perform grounding for the same predicate multiple times. However, the problem is that lifted inference can only be used to learn the probability for restricted rule sets. There is a lot of scope for developing systems that can incorporate lifted inference in the probabilistic rule learning process so that these systems can learn complex rules within a shorter period of time.

Table 2. Features Supported by the Probabilistic Rule Learning Systems

Learning System	**Supports Learning from Probabilistic Data**	**Employs Background Knowledge**	**Learns Rules for Multiple Target Predicates**	**Requires Language Bias**	**Supports Predicates with Any Number of Arguments**	**Learns Complex Rules**
Aleph	No	Yes	No	Yes	Yes	Yes
cplint	Yes	Yes	Yes	Yes	Yes	Yes
Neural LP	No	No	Yes	No	No [*]	No [**]
LIME-Aleph	No	Yes	No	Yes	Yes	Yes

Note: [*] uses only binary predicates, and [**] learns only chain rules.

Learn complex rules using sub-symbolic systems. Sub-symbolic systems like Neural LP are promising because of their efficiency in handling large and noisy datasets. These systems are also computationally efficient, but they mostly learn chain-like rules. Hence, there is scope for research in the space of sub-symbolic systems that they can also learn complex rules.

Build hybrid systems for structure learning. Although hybrid systems combine symbolic and sub-symbolic representations, they do not learn a single representation collaboratively. To date, DeepProbLog [15] is the only system that truly integrates symbolic and sub-symbolic representations for a single task. In DeepProbLog (as discussed in Section 5.3), a sub-task of the learning problem is completed by using a sub-symbolic representation and the task is then finalized on the symbolic level by using the result of the sub-task.

Data efficient learning. In general, a symbolic system learns rules from a small amount of data, whereas a sub-symbolic system requires a large amount of data. However, sub-symbolic systems can efficiently handle large and noisy data in contrast to symbolic systems. Effective data handling techniques can be introduced in a symbolic system [21], and techniques to learn from a small amount of data can be used in a sub-symbolic system [42].

Use of background knowledge. As we have seen, the symbolic systems easily incorporate problem specific information in the form of background knowledge for learning rules. However, it is a challenging task to integrate the background knowledge in a sub-symbolic system because of its internal representation [36, 44]. Furthermore, it is an open research question how commonsense knowledge can be used for direct inference in sub-symbolic learning systems.

Table 2 compares the features of the surveyed probabilistic rule learning systems and illustrates their characteristics.

These features suggest the following recommendations[6]:

- If we want to learn probabilistic rules from probabilistic data, then we can use *cplint*, as Aleph and Neural LP cannot handle probabilistic data.
- Background knowledge plays an important role for most problem domains where we need to represent domain-specific information. If we want to learn probabilistic rules employing the background knowledge, then Aleph and *cplint* can be used.
- If we want to learn rules with multiple target predicates—that is, rules with different predicates in the head—then we can employ *cplint* or Neural LP.

[6]We have excluded the LIME-Aleph system in the summary, since it uses Aleph for rule learning and Aleph is already in the list.

- We can use Aleph and *cplint* to learn complex rules where the body contains any number of predicates in any sequence unlike a chain rule. Aleph and *cplint* also support predicates with any number of arguments. However, Neural LP only learns chain rules and only supports binary predicates.
- Only Neural LP does not require a setting for the language bias. As a result, Neural LP demands less configuration than the other systems discussed in this survey.

7 CONCLUSION

Research on probabilistic rule learning is gaining a lot of attention because it can deal with uncertainty in a given domain, combine high level representations with efficient (probabilistic) reasoning, and make the reasoning process transparent.

We have reviewed several probabilistic rule learning systems that use different approaches to formalize a problem description and discussed the architectures of these systems. The presented approaches usually only need a small amount of example instances, and the background knowledge is declarative and can be incorporated in an easy way. These systems have been used effectively in the domain of knowledge-based reasoning [5], bioinformatics [8], financial analysis [10], and web data analysis [2].

It is our hope that this survey is informative for practitioners who want to explore the potential of this novel technology and for researchers who want to tackle one of the open research questions in the exciting field of probabilistic rule learning.

ACKNOWLEDGMENT

This work was funded by the International Macquarie Research Excellence Scholarship (iMQRES), Macquarie University, Sydney, Australia.

REFERENCES

[1] Elena Bellodi and Fabrizio Riguzzi. 2013. Expectation maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis* 17, 2 (2013), 343–363.

[2] Elena Bellodi and Fabrizio Riguzzi. 2015. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming* 15, 2 (2015), 169–212.

[3] William W. Cohen. 2016. TensorLog: A differentiable deductive database. arXiv:1605.06523.

[4] Andrew Cropper and Stephen H. Muggleton. 2016. Metagol system. Retrieved May 25, 2020 from https://github.com/metagol/metagol.

[5] Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. 2015. Inducing probabilistic relational rules from probabilistic examples. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15).*

[6] Luc De Raedt and Kristian Kersting. 2008. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming: Theory and Applications*, Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton (Eds.). Vol. 4911. Springer, Berlin, Germany, 1–27.

[7] Luc De Raedt and Angelika Kimmig. 2015. Probabilistic (logic) programming concepts. *Machine Learning* 100, 1 (2015), 5–47.

[8] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. Problog2: Probabilistic logic programming. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* 312–315.

[9] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning libraries of subroutines for neurally–guided Bayesian program induction. In *Advances in Neural Information Processing Systems.* 7805–7815.

[10] Arnaud Nguembang Fadja and Fabrizio Riguzzi. 2019. Lifted discriminative learning of probabilistic logic programs. *Machine Learning* 108, 7 (2019), 1111–1135.

[11] Artur d'Avila Garcez, Marco Gori, Luis C. Lamb, Luciano Serafini, Michael Spranger, and Son N. Tran. 2019. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. arXiv:1905.06088.

[12] Arcchit Jain, Tal Friedman, Ondrej Kuzelka, Guy Van den Broeck, and Luc De Raedt. 2019. Scalable rule learning in probabilistic knowledge bases. In *Proceedings of the 1st Conference on Automated Knowledge Base Construction (AKBC'19)*.

[13] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. In *Proceedings of the International Conference on Learning Representations*.

[14] Muhammad Taimoor Khan and Howard Shrobe. 2019. Security of cyberphysical systems: Chaining induction and deduction. *Computer* 52, 7 (2019), 72–75.

[15] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. DeepProbLog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*. 3749–3759.

[16] Eric McCreath and Arun Sharma. 1995. Extraction of meta-knowledge to restrict the hypothesis space for ILP systems. In *Proceedings of the 8th Australian Joint Conference on Artificial Intelligence*. 75–82.

[17] Jack Minker. 1982. On indefinite databases and the closed world assumption. In *Proceedings of the International Conference on Automated Deduction*. 292–308.

[18] Stephen Muggleton. 1991. Inductive logic programming. *New Generation Computing* 8, 4 (1991), 295–318.

[19] Stephen Muggleton. 1995. Inverse entailment and Progol. *New Generation Computing* 13, 3–4 (1995), 245–286.

[20] Stephen Muggleton. 1996. Learning from positive data. In *Proceedings of the International Conference on Inductive Logic Programming*. 358–376.

[21] Stephen Muggleton, Wang-Zhou Dai, Claude Sammut, Alireza Tamaddoni-Nezhad, Jing Wen, and Zhi-Hua Zhou. 2018. Meta-interpretive learning from noisy images. *Machine Learning* 107, 7 (2018), 1097–1118.

[22] Stephen Muggleton and Luc De Raedt. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19 (1994), 629–679.

[23] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. 2015. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning* 100, 1 (2015), 49–73.

[24] Stephen H. Muggleton, Ute Schmid, Christina Zeller, Alireza Tamaddoni-Nezhad, and Tarek Besold. 2018. Ultra-strong machine learning: Comprehensibility of programs learned with ILP. *Machine Learning* 107, 7 (2018), 1119–1140.

[25] David Poole. 2003. First-order probabilistic inference. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 3. 985–991.

[26] J. Ross Quinlan. 1990. Learning logical definitions from relations. *Machine Learning* 5, 3 (1990), 239–266.

[27] Johannes Rabold, Hannah Deininger, Michael Siebers, and Ute Schmid. 2019. Enriching visual with verbal explanations for relational concepts—Combining LIME with Aleph. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 180–192.

[28] Luc De Raedt, Sebastijan Dumancic, Robin Manhaeve, and Giuseppe Marra. 2020. From statistical relational to neuro-symbolic artificial intelligence. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI'20)*. 4943–4950. DOI: https://doi.org/10.24963/ijcai.2020/688

[29] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan & Claypool.

[30] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should I trust you? Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 1135–1144.

[31] Fabrizio Riguzzi. 2018. *Foundations of Probabilistic Logic Programming*. River Publishers.

[32] Fabrizio Riguzzi and Damiano Azzolini. 2020. *cplint* Documentation. *SWI-Prolog Version.* Retrieved May 25, 2020 from http://friguzzi.github.io/cplint/_build/latex/cplint.pdf.

[33] Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese. 2014. A history of probabilistic inductive logic programming. *Frontiers in Robotics and AI* 1 (2014), 6.

[34] Fabrizio Riguzzi and Theresa Swift. 2018. A survey of probabilistic logic programming. In *Declarative Logic Programming: Theory, Systems, and Applications*. ACM Books, 185–228.

[35] Tim Rocktäschel and Sebastian Riedel. 2017. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*. 3788–3800.

[36] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. 2015. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1119–1129.

[37] Taisuke Sato. 1995. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*. 715–729.

[38] Taisuke Sato and Yoshitaka Kameya. 1997. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 97. 1330–1339.

[39] Maria I. Sessa. 2002. Approximate reasoning by similarity-based SLD resolution. *Theoretical Computer Science* 275, 1–2 (2002), 389–426.
[40] Ashwin Srinivasan. 2007. The Aleph Manual. Retrieved May 25, 2020 from http://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html.
[41] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. 2004. Logic programs with annotated disjunctions. In *Proceedings of the International Conference on Logic Programming*. 431–445.
[42] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys* 53, 3 (2020), 1–34.
[43] Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. 2019. NLProlog: Reasoning with weak unification for question answering in natural language. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 6151–6161.
[44] Dirk Weissenborn, Tomáš Kočiskỳ, and Chris Dyer. 2017. Dynamic integration of background knowledge in neural NLU systems. arXiv:1706.02596.
[45] Fan Yang, Zhilin Yang, and William W. Cohen. 2017. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*. 2319–2328.

Received May 2020; revised November 2020; accepted January 2021