Logic Programming for Knowledge Graph Completion

Damiano Azzolini[1], Matteo Bonato[2], Elisabetta Gentili[2,*] and Fabrizio Riguzzi[3]

[1]*Department of Environmental and Prevention Sciences – University of Ferrara, Ferrara, Italy*
[2]*Department of Engineering – University of Ferrara, Ferrara, Italy*
[3]*Department of Mathematics and Computer Science – University of Ferrara, Ferrara, Italy*

Abstract
A knowledge graph (KG) represents a domain of interest with a graph where some of the involved entities are linked with an edge. Knowledge Graph Completion (KGC) is a well-known task for KGs which requires finding missing connections. KGC has been studied for many years with multiple solutions available based on both symbolic and sub-symbolic techniques. In this paper, we would like to answer the question: can parameter learning for Probabilistic Logic Programming be a competitive algorithm to solve the KGC task? An empirical evaluation on the most common KGC datasets allows us to provide a negative answer to such a question.

Keywords
Knowledge Graph Completion, Parameter Learning, Statistical Relational Artificial Intelligence, Logic Programming

1. Introduction

Large Knowledge Graphs (KGs) pose interesting challenges, especially in extracting information from them. One of them goes under the name of Knowledge Graph Completion (KGC): given a domain represented with a graph, the goal is to find missing edges among the involved entities. There are many existing solutions to solve this task, mainly based on deep learning models [1]. However, despite having huge success in solving KGC, they are considered black boxes, since the predictions cannot be explained to the user.

Probabilistic logic languages [2], and in particular Probabilistic Logic Programming (PLP), are interesting formalisms to express uncertainty with a logic program extended with probabilistic atoms and rules. These are inherently explainable, since they represent information with easily interpretable logical rules. Recently, the authors of [3] proposed to learn logical rules from a KG and associate a numerical parameter to them. Motivated by their success, in this paper, we cast the KGC problem as a parameter learning problem in PLP: given a set of probabilistic logical rules, the goal is to associate probabilities to them such that the probabilities of the provided examples are maximized. For KGC, the examples are the edges existing in the KG. This can be solved, for example, with Expectation Maximization. We tested multiple datasets with different configurations and different ways to generate rules, to check whether the proposed approach is competitive with state-of-the-art solutions. Unfortunately, we found that this is not the case. However, this provides an interesting direction for future improvement of existing algorithms for parameter learning in PLP.

The paper is structured as follows: Section 2 introduces the background notions; in Section 3 we describe the approaches used for generating the rules; Section 4 shows the results of the experiments; in Section 5 we discuss related work; lastly, in Section 6 we draw the conclusions and propose future work directions.

CILC 2024: 39th Italian Conference on Computational Logic, June 26-28, 2024, Rome, Italy
*Corresponding author.
damiano.azzolini@unife.it (D. Azzolini); matteo01.bonato@edu.unife.it (M. Bonato); elisabetta.gentili1@unife.it (E. Gentili); fabrizio.riguzzi@unife.it (F. Riguzzi)
0000-0002-7133-2673 (D. Azzolini); 0009-0006-6901-0540 (E. Gentili); 0000-0003-1654-9703 (F. Riguzzi)

2. Background

KGs are graph-based representations of knowledge in terms of relationships between entities. A KG can be represented as a set of triples (h, r, t) where *r* is the relation, and *h* and *t* are the head (start) and tail (end) entities of the relation, respectively.

Real-world KGs are usually incomplete and sparse, thus inference of missing information (entities or relationships) is often required. This task is referred to as KGC. Depending on the information to be inferred, KGC can be divided into specific tasks [4], such as link prediction, entity prediction, or relation prediction. More formally, link prediction is the task of predicting either the tail t of a triple $(h, r, ?)$, or the head h of a triple $(?, r, t)$.

Given a completion task $(h, r, ?)$ on a graph $\mathbb{G}$, the goal is to find an entity t such that $(h, r, t) \notin \mathbb{G}$ is true. Candidate triples are associated with a score and they are sorted in descending order. Given an answer t, its rank $rank(t)$ is the position in the list of answers. Special care has to be taken in case there is a group with several answers with the same score. In this case, there are various approaches for computing rank(t). For example, minimum and maximum ranking take respectively the lowest and the highest rank in the group, while average ranking assigns instead the average rank of the group. In our implementation, we always consider the average ranking, even if other tested tools may adopt different strategies.

The graph $\mathbb{G}$ is often split into three datasets, training set T_{train}, test set T_{test}, and validation set T_{valid}, used to train the model and evaluate its performance. Given a set of test triples T_{test}, KGC algorithms are usually evaluated on link prediction tasks in terms of the following metrics:

- Mean Rank (MR), which is the average rank of the test triples:

$$MR = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} rank(t)$$

- Mean Reciprocal Rank (MRR), which is the average reciprocal individual rank of the test triples:

$$MRR = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} \frac{1}{rank(t)}$$

- Hits@K, which represents the proportion of the test triples ranked in the top K positions:

$$Hits@K = \frac{|\{t \in T_{test} \mid rank(t) \leq K\}|}{|T_{test}|}$$

Often, the ranking for a test triple (h, r, t) includes a candidate t' such that $(h, r, t') \notin T_{test}$. Even if t' is not the correct prediction, a triple (h, r, t') may appear in the training or validation set. To not penalize the ranking of the correct candidate t, predictions that appear in triples of the training or validation set can be filtered out. By doing so, a filtered version of the metrics will be computed [5].

2.1. AnyBURL

AnyBURL [3] is an anytime algorithm used to learn rules from KGs by following the bottom-up paradigm. AnyBURL iteratively samples from a KG $\mathbb{G}$ random paths of length *n*, where *n* is the number of edges and starts from $n = 2$. These paths are then assembled into ground logical rules where the first edge is considered as the head and the remaining as the body.

Given a set of triples $\{(e_0, h_0, e_1), (e_1, b_1, e_2), \ldots, (e_n, b_n, e_{n+1})\}$, we can construct a ground path rule R of the form $h(e_0, e_1) \leftarrow b_1(e_1, e_2), \ldots, b_n(e_n, e_{n+1})$. AnyBURL extracts three types of rules (listed in Table 1): i) rules that generalize acyclic ground path rules, i.e., rules where $e_0 \neq e_{n+1}$ (called **AC2**); ii) rules that generalize cyclic ground path rules, i.e., rules where $e_0 = e_{n+1}$ (called **C**); iii) rules that generalize both acyclic and cyclic ground path rules (called **AC1**). This process continues until a certain saturation parameter (which depends on the contribution to the overall performance of the

Table 1
Rules extracted by AnyBURL from paths on KGs. X and Y are variables that appear in the head, while A_i can appear only in the body. Lowercase letters indicate constants.

Name	Structure
C	$h(Y, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, Y)$
AC1	$h(e_0, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, e_{n+1})$
AC2	$h(e_0, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, A_{n+1})$

considered rules) is reached. At each iteration, the length of the considered path is increased by 1. The score of a rule R is the confidence, computed as follows [6]:

$$support(R) = |\{\theta_{XY} \mid \exists \theta_Z R_b \theta_{XYZ} \wedge R_h \theta_{XY}\}|$$

$$conf(R) = \frac{support(R)}{|\{\theta_{XY} \mid \exists \theta_Z R_b \theta_{XYZ}\}|}$$

where θ_{XY} is a grounding for variables X and Y appearing in the head R_h of R, θ_Z is a grounding for the variables appearing in the body R_b of R other than X and Y, and θ_{XYZ} is the union of θ_{XY} and θ_Z.

To score entities, AnyBURL adopts the maximum aggregation, in which candidate entities e_i are ordered according to the maximum confidence of all the rules R_i that generated them, i.e.,

$$score(e_i) = max\{conf(R_1), \ldots, conf(R_n)\}$$

where e_i is the entity and R_i are the rules that predicted it.

When the number of rules is high, computing KGC metrics is very expensive. To overcome this, AnyBURL exploits multiple threads and limits the search for candidates if during the grounding of a rule a branch with more than 10^4 children is found [7]. For example, consider the rule

$$hasGender(X, Y) \leftarrow bornIn(X, A), bornIn(B, A), hasGender(B, Y)$$

and the query $hasGender(susi, T)$ where $susi$ is born in the USA. The second body atom unifies B with each person known to be born in the USA, so there can be more than 10^4 instantiations. In this case, AbyBURL only retrieves the first 10^4. There is also a parameter TOP_K_OUTPUT that indicates the maximum number of answers to consider for KGC queries. In fact, for example, when computing Hits@10, it is not necessary to compute all answers. Usually, this parameter is set to 100. However, this only provides an approximation of the metrics because the answers with a score below that of the correct answer do not contribute to the rank of it. So even if we first find the correct answer and 99 answers with a lower score, we cannot be sure that the rank of the correct answer is 1 as there can be more answers not yet computed with a larger score. However, it provides a good trade-off between speed and precision. There are also highly optimized libraries for metrics computations such as PyClause [8], which also offers an interface for AnyBURL.

2.2. SAFRAN

One drawback of the maximum aggregation is that it considers only a single rule instead of considering a weighted combination of all the rules that predict a certain entity. Noisy-Or aggregation tries to address this by assuming that the confidence of the rule is a probability and by considering the Noisy-Or of the predictions:

$$score(e) = 1 - \prod_{i=1}^{k}(1 - conf(R_i))$$

However, Noisy-Or aggregation does not account for rules redundancy, which often occurs in real-world KGs, resulting in worse performance than Maximum aggregation due to an overestimation of the confidence because of double counting [9]. The authors of [9] tried to overcome this issue and proposed SAFRAN, a framework that adopts a novel aggregation approach called *non-redundant Noisy-Or*. This approach clusters redundant rules based on their redundancy degree; then, it applies maximum aggregations to the rules in the same clusters; lastly, it aggregates predictions of different clusters with Noisy-Or. Two rules R_i, R_j are assigned to the same cluster if their redundancy degree is higher than a certain threshold, that is $sim(R_i, R_j) > th$, where $sim(R_i, R_j)$ is the Jaccard Index of the sets of inferred triples and th is a threshold. To find the optimal value for th, which depends on the relation and rule type, SAFRAN adopts grid search or random search.

2.3. LIFTCOVER+

PLP combines logic-based languages and uncertainty [2]. Thanks to its expressiveness, PLP under the distribution semantics [10] has been adopted in many domains where uncertainty is relevant [11, 12, 13]. Logic Programs with Annotated Disjunctions (LPADs) [14] are a PLP language under the distribution semantics. In LPADs, heads of clauses are disjunctions in which each atom is annotated with a probability. Liftable Probabilistic Logic Programs [15] are a restriction of probabilistic logic programs in which inference can be performed in a lifted way [16] by considering populations of individuals instead of each individual separately. Liftable PLP programs contain clauses with a single annotated atom in the head and the predicate of this atom is the same for all clauses:

$$h_i : \Pi_i :- b_{i1}, \ldots, b_{iu_i}$$

where the single atom in the head is built over predicate *target/a*, which is the target of learning, and where a is the arity. Bodies of the clauses may contain other predicates than *target/a*, called *input predicates*, that are certain, meaning that their facts and rules have a single atom in the head with probability 1. To compute the probability of a query *q*, it is necessary to find only the number of ground instantiations of clauses so that the body is true and the head is equal to *q*. If clause C_i has m_i instantiations, $\{\theta_{i1}, \ldots, \theta_{im_i}\}$, every θ_{ij} corresponds to a random variable X_{ij} and $P(X_{ij} = 1) = \Pi_i$, while $P(X_{ij} = 0) = 1 - \Pi_i$. A query *q* is true if at least one random variable for a rule is true, and is false only if none of the random variables is true. All the random variables are mutually independent, so $P(q) = 1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}$.

Given a set $E^+ = \{e_1, \ldots, e_Q\}$ of positive examples (i.e., a set of atoms), a set $E^- = \{e_{Q+1}, \ldots, e_R\}$ of negative examples (i.e., a set of atoms), and a background knowledge B defining the input predicates, LIFTCOVER learns the parameters of the clauses using either Expectation-Maximization (EM) or Limited-memory BFGS (LBFGS). The likelihood L of the examples can be unfolded as

$$L = \prod_{l=1}^{n}(1 - \Pi_l)^{m_{l-}} \prod_{q=1}^{Q}\left(1 - \prod_{l=1}^{n}(1 - \Pi_l)^{m_{lq}}\right)$$

where m_{iq} (m_{ir}) is the number of instantiations of C_i whose head is e_q (e_r) and whose body is true, and $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$. The gradient of the likelihood is computed as:

$$\frac{\partial L}{\partial \Pi_i} = \frac{L}{1 - \Pi_i}\left(\sum_{q=1}^{Q} m_{iq}\left(\frac{1}{P(e_q)} - 1\right) - m_{i-}\right)$$

The equation $\frac{\partial L}{\partial \Pi_i} = 0$ does not admit a closed-form solution, thus optimization is needed to find the maximum of L. Finally, the clauses with a probability below a user-defined threshold are discarded.

The EM algorithm [17] is used to find the maximum likelihood estimates of parameters. First, in the Expectation step, the distribution of the unseen variables in each instance is computed given the observed data and the current value of the parameters. Then, during the Maximization step, the new parameters are computed so that the likelihood is maximized. The algorithm repeats these two steps until there are no more improvements in the likelihood. To use the EM algorithm, the distribution of the hidden variables given the observed ones, $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$, has to be computed. Given that $P(X_{ij} = 1, e) = P(e|X_{ij} = 1) \cdot P(X_{ij} = 1) = P(X_{ij} = 1) = \Pi_i$ since $P(e|X_{ij} = 1) = 1$,

$$P(X_{ij} = 1|e) = \frac{P(X_{ij} = 1, e)}{P(e)} = \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}}$$

$$P(X_{ij} = 0|e) = 1 - \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}}$$

Since $P(X_{ij} = 1, \neg e) = P(\neg e|X_{ij} = 1) \cdot P(X_{ij} = 1) = 0$ and $P(\neg e|X_{ij} = 1) = 0$,

$$P(X_{ij} = 1|\neg e) = 0$$

$$P(X_{ij} = 0|\neg e) = 1$$

LIFTCOVER+ [18, 19] is a modified version of LIFTCOVER that learns the parameters and the structure of liftable probabilistic logic programs using regularization to prevent overfitting. Moreover, it replaces LBFGS with gradient descent to optimize the likelihood.

Regularization is a widely used strategy to inhibit overfitting, by introducing a penalty term in the loss function to penalize large weights. Since clauses with small weights are removed because they have little influence on the probability of the query, fewer clauses with large weights, i.e., a smaller theory, should be obtained. Bayesian, L1, or L2 regularization can be used in the Maximization step of the EM algorithm. The main difference in L1 and L2 regularization lies in the penalty introduced in the loss function. The L1 penalty term is the sum of the absolute values of the parameters, while the L2 penalty term is the sum of their squares. Furthermore, L1 tries to bring the parameters close to 0 to create a sparse model, i.e., a model with few non-zero parameters. However, L1 is computationally inefficient. On the other hand, L2 is efficient but produces non-sparse solutions. In both cases, the impact of regularization is controlled by a regularization coefficient: the higher it is, the stronger the regularization will be.

The L1 objective function [20] is:

$$J_1(\theta) = N_1 \cdot log\theta + N_0 \cdot log(1 - \theta) - \gamma\theta$$

while the L2 objective function [20] is:

$$J_2(\theta) = N_1 \cdot log\theta + N_0 \cdot log(1 - \theta) - \frac{\gamma}{2}\theta^2$$

where $\theta = \pi_i$, N_0 and N_1 are the expected occurrences of $X_{ij} = 0$ and $X_{ij} = 1$ respectively computed during the Expectation step, and γ is the regularization coefficient.

The value of θ that maximizes the objective function is computed in the Maximization step by solving the equation $\frac{\partial J(\theta)}{\partial\theta} = 0$ [20]. $J_1(\theta)$ is maximum at:

$$\theta_1 = \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0 + N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})}$$

while $J_2(\theta)$ is maximum at:

$$\theta_2 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}} \cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right)}{3N_0+3N_1+\gamma}\right)}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3}$$

In Bayesian regularization, parameters are updated assuming a prior distribution in the form of a Dirichlet probability density with parameters $[a, b]$, where *a* and *b* are the number of extra occurrences observed of $X_{ij} = 1$ and $X_{ij} = 0$ respectively. Parameters are shrunk when *b* is much larger than *a*.

To apply EM, we need to store the values m_{l-} for each rule C_l and the values m_{lq} for each clause C_l and each positive example e_q. We can store these values with vector M_- and matrix M_+ respectively, of size n the first and $n \times Q$ the latter. We have implemented the operations to be performed in the Expectation and Maximization phases as vector/matrix operations on M_- and M_+. LIFTCOVER+ now offers implementations of the algorithm in Prolog and in Python using the Janus interface [21, 22]. The Python version can exploit either the CPU with package numpy or the GPU with package cupy.

3. Approaches for Rule Generation

In this section, we describe the solutions we considered to generate rules whose probabilities are learnt via LIFTCOVER+.

3.1. Generating Cyclic Rules by Computing Paths in the KG

Here we generate only cyclic rules by generating tuples of relations $r_0, r_1, \ldots,$ and then converting them to rules. The tuples of relations have $n + 1$ elements where the first element is the relation that goes into the head of the clause and the other n are those that go in the body, thus obtaining a body with n atoms. For this phase, each triple (h, rel, t) in the dataset is translated into an atom $t(h, rel, t)$. For example, the tuple `(r0,r1,r2,r3)` is translated into the rule

```
r(A,r0,B):-r(A,r1,C),r(C,r2,D),r(D,r3,B).
```

where $r/3$ is defined as

```
r(S,R,T):-
  t(S,R,T).

r(S,i(R),T):-
  t(T,R,S).
```

so that both the relation (`R`) and their inverse (`i(R)`) are considered. The tuples of relations are obtained by starting from triples in the training set. For example, if we want to extract 4 relations we look for values of `R1,R2,R3,R4` such that the query

```
r(h,R1,C),r(C,R2,D),r(D,R3,E),r(E,R4,t).
```

succeeds at least once. Then duplicate tuples are removed.

Removing duplicates can be very expensive, so we looked for different ways to do it. In the following, we consider SWI Prolog [23] as inference engine. We tried with tabling [24], which has the property of storing each answer only once, but it was faster to directly use tries[1]. Moreover, we exploited parallelism in order to speed up the computation. In particular, we used the SWI library predicate `concurrent_maplist(Goal,List)` that applies `Goal` to each element of `List` in parallel using threads. We also wrote a helper predicate `chunks(List,N,SubLists)` that splits the elements of `List` into `N` lists of approximately equal length and returns them in `SubLists`. For example, to generate cyclic rules with 3 atoms in the body that contain the training tuples, we used the code below:

```
main:-
  tell('tuples3.pl'),
  rels(Rels),
  chunks(Rels,32,Chunks),
```

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

```
  trie_new(Tr),
  concurrent_maplist(genpaths3(Tr),Chunks),
  trie_gen(Tr,q(R,R1,R2,R3)),
  write('(tt(A,'), write_canonical(R), write(',B):0.5 :- r(A,'),
  write_canonical(R1), write(',C), r(C,'),
  write_canonical(R2), write(',D), r(D,'), write_canonical(R3),
  writeln(',B)),'),
  fail.

main:-
   told.

genpaths3(Tr,Rels):-
  member(R,Rels),
  path3(R,R1,R2,R3),
  trie_insert(Tr,q(R,R1,R2,R3)),
  fail.

genpaths3(_,_).

path3(R,R1,R2,R3):-
    r(S,R,T), r(S,R1,I), T\=I,
    r(I,R2,J), r(J,R3,T), J\=S.
```

where $rels/1$ is a predicate that computes the list of all possible relations. These are then split into 32 chunks and each chunk is processed in parallel by a separate thread. Since operations on tries are thread-safe, parallelization does not pose a problem. The reason for the tests `T\=I` and `J\=S` is to avoid generating rules such as

```
r(S,r,T):-r(S,r,T),r(T,r1,J),r(J,i(r1),T).
r(S,r,T):-r(S,r1,I),r(T,i(r1),S),r(S,r,T).
```

This approach allows a quick sampling of a fraction f of the tuples by generating a random number X in [0,1] and checking whether $X < f$, as in, for example,

```
trie_gen(Tr,q(R,R1,R2,R3)), random(X), X < 0.05,
```

where we sample 5% of the tuples.

3.2. Generating Rules as in AnyBURL

With this approach, for each relation in the KG, we find the lists of start (head) and end (tail) nodes. Then, we find a user-defined number of paths starting from each start (end) node up to a maximum length also specified by the user. These are used to generate **AC1**, **AC2**, and **C** rules. Here as well duplicates are removed and each rule is associated with an initial probability value of 10^{-4}. This set of rules is passed as input theory to LIFTCOVER+ to tune the probabilities. The rankings of the candidates for an entity are computed via Noisy-OR aggregation.

3.3. Generating Rules by Weighted Sampling

We tried another approach to generate rules: for each dataset, we extract all the relations together with their number of occurrences. Call the list with all the relations L_r. We associate each relation with a probability proportional to its occurrences. To generate a rule, we proceed as follows: we first sample a relation from L_r to consider as head relation. Then, we select a random number between two

Table 2
Description of the datasets used for the experiments. For each dataset, we report the number of triples in the training, validation, test sets, and total, and the number of relations and entities.

Dataset	Train	Valid	Test	Total	Entities	Relations
Nations	1592	199	201	1992	28	55
WN18RR	86835	3034	3134	93003	71453	11
FB15k-237	272115	17535	20466	310116	14505	237
Nell	228426	129810	144307	502543	63361	400

and four and sample again the same number of relations from L_r, that will be considered for the body. Both sampling phases take into account the probability associated with each relation. Starting from the selected relations, we generate rules by creating an atom with two variables for each relation and then connect the atoms to create a chain rule. To clarify, suppose we have $r0$ for the head and $r1$ and $r2$ for the body. We obtain the rule `r(A,r0,B) :- r(B,r1,C), r(C,r2,D)`. We repeat this process for a fixed number of rules. Once we have all the rules, we compute the cumulative number of instantiations for all the bodies, call it n_i, and associate each rule with a score given by the ratio between its number of body instantiations and n_i. This method is implemented in Prolog as well.

4. Experimental Evaluation

In this section, we report the experimental evaluation of our approach on well-known KGC datasets.

4.1. Datasets

In our experiments, we consider the following datasets: FB15K-237 [25] which contains entity pairs extracted from FreeBase, WN18RR [26] which contains WordNet entities and explicit relations, NELL [27] which was built with an agent (called Never-Ending Language Learner) that reads the web and learns over time, and Nations [28] which contains data about economic, diplomatic, military, and social dyadic interactions among a reduced set of 182 nation-dyads in the years between 1950 and 1965. The number of tuples, relations, and entities for each dataset are listed in Table 2.

4.2. Setup Parameter Learning with LIFTCOVER+

We learn the parameters of the rules using the EM algorithm of LIFTCOVER+ because it gives solutions with better quality than gradient descent and LBFGS, and run it on GPUs to speed up computations. However, the need to compute and store the matrix M_+ for the positive examples (see Section 2.3) requires limiting the number of positive examples and rules. In fact, with n rules, Q positive examples, if we assume that an integer requires 32 bits (4 bytes) of storage, storing the matrix M_+ requires $n \cdot Q \cdot 4$ bytes. For example, with 10^5 rules and 10^5 positive examples, we need 40GB of memory, close to the limit of the available hardware that is available to us (64GB). Thus we need to subsample examples for the FB15k-237 and NELL datasets, since they have an excessive number of examples. In particular, we sampled 81,196 positive examples for FB15k-237 and 4,670 for NELL. Moreover, we randomly generated 10^5 negative examples for FB15k-237, $2 \cdot 10^5$ for NELL, $2 \cdot 10^5$ for WN18RR, and $5 \cdot 10^3$ for Nations by repeatedly randomly sampling a relation r, a head h, and a tail t and checking whether the triple (h, r, t) is not present in the KG until the required number of examples is obtained.

4.3. Metrics Computation

We implemented in Prolog an algorithm for computing the Hits@1, Hits@3, Hits@5, Hits@10, and MRR metrics. Algorithm 1 shows the pseudocode of function METRICS that, given the test set and the

Algorithm 1 Algorithm for computing metrics.

```
function METRICS($TestSet$, $TrainingSet$, $ValidationSet$, $Theory$)
    $Ranks = \emptyset$
    for all $r(h, r, t) \in TestSet$ do
        $SortedAns \leftarrow$ COMPUTEANS($r(h, r, t)$, $Theory$, $TrainingSet$, $ValidationSet$)
        $rank \leftarrow$ COMPUTERANK($t$, $SortedAns$)
        $Ranks \leftarrow Ranks \cup \{rank\}$
    end for
    return COMPUTEHITSMRR($Ranks$)
end function
function COMPUTEANS($r(h, r, t)$, $Theory$)
    $ProbCorrAns \leftarrow$ COMPUTEPROB($r(h, r, t)$, $Theory$)
    $Inst \leftarrow$ FINDINSTANTIATIONS($r(h, r, T)$, $Theory$)
    $FilteredInst \leftarrow$ FILTER($Inst$, $TrainingSet$, $ValidationSet$)
    $Answers \leftarrow \{(ProbCorrAns, t)\}$
    $N = 1$
    for all $r(h, r, t') \in FilteredInst$ do
        $Prob \leftarrow$ COMPUTEPROB($r(h, r, t')$, $Theory$)
        if $Prob \geq ProbCorrAns$ then
            $Answers \leftarrow Answers \cup \{(Prob, t')\}$
            $N \leftarrow N + 1$
        end if
        if $N \geq 20$ then
            return SORT($Answers$)
        end if
    end for
end function
```

set of rules, returns the metrics. The function, for each triple $t(h, r, t)$ in the test set, calls function COMPUTEANS($r(h, r, t), Theory$) that returns a set of pairs (P, t') for answers t' to $r(h, r, T)$. COMPUTEANS first computes the probability of the correct answer t. Then, it finds all possible instantiations of the query $r(h, r, T)$ using the Prolog code

```
setof(r(h,r,T),Prog^find_inst(Prog,r(h,r,T)),Atoms)
```

where `find_inst/2` is defined as

```
find_inst(Prog,Ex):-
  member(((Head : _P) :- Body),Prog),
  Head = Ex, Body.
```

After this, COMPUTEANS removes the instantiations present in the training or validation set and cycles over the remaining ones, computing their probability and keeping a counter N of the number of answers with a probability greater or equal to $ProbCorrAns$, the probability of the correct answer. If the probability $Prob$ of an instantiation (h, r, t') is greater or equal to $ProbCorrAns$, then the pair $(Prob, t')$ is added to the current set of scored answers, and the counter N is updated. In fact, if $Prob < ProbCorrAns$, t' need not be stored, as it does not influence the ranking of t. Then we test N: if it is 20 or more, we can stop examining instantiations, as the rank of t would be for sure greater than 10. In fact, the worst case is that there are 20 answers all with the same probability: in that case, the rank of each answer would be $\frac{20+1}{2} = 10.5$ according to the average approach to ranking computation.

Function METRICS computes exact values for Hits@1, Hits@3, Hits@5, and Hits@10, while a pessimistic approximation to MRR, as the actual rank of a correct answer could be lower, while we stop as soon as we have found 19 other answers with a probability greater or equal to that of the correct

Table 3
Maximum length of the body of rules and number of rules for each dataset and each rule generation method. Paths denotes the approach described in Section 3.1.

Dataset	Rule Generation Method	Max Length	#Rules
Nations	Paths	3	305,365
	AC1 + AC2 + C	3	2,243
	Weighted Sampling	4	10^5
WN18RR	Paths	3	3,076
	Weighted Sampling	4	10^5
FB15K-237	Paths	3	33,696
	Weighted Sampling	4	10^5
Nell	Paths	3	101,242
	Weighted Sampling	4	10^5

Table 4
For each dataset and algorithm, we report the metrics Hits@1, Hits@3, Hits@5, Hits@10 and MRR together with the time taken by parameter learning. For Paths+Conf we used the confidence as the parameters of the rules. We do not report the configurations that were not able to solve the task.

Dataset	Approach	H@1	H@3	H@5	H@10	MRR	Learning Time (s)
Nations	Paths+EM	0.4577	0.7861	0.8607	0.9950	0.6389	467.41
	Paths+Conf	0.4926	0.7910	0.8557	0.9701	0.6586	-
	AC1+AC2+C+EM	0.2189	0.5721	0.7612	0.9104	0.4404	6.69
	Weighted Sampling	0.5075	0.7811	0.8458	0.9652	0.6472	-
WN18RR	Paths+EM	0.3405	0.4537	0.4936	0.5378	0.4193	189.74
	Paths+Conf	0.4371	0.5057	0.5370	0.5731	0.4894	-
	Weighted Sampling	0.0306	0.0826	0.1107	0.1512	0.0741	-
FB15K-237	Paths+EM	0.2143	0.3093	0.3530	0.4263	0.2844	936.36
	Paths+Conf	0.2448	0.3528	0.4034	0.4801	0.3232	-
Nell	Paths+EM	0.0004	0.0018	0.0028	0.0053	0.0019	19,713.04
	Paths+Conf	0.0010	0.0025	0.0038	0.0067	0.0033	-

answer. Since the values of the hits metrics are exact, this function differs from the one of AnyBURL (also implemented in PyClause). Note that function Metrics can also be parallelized by executing concurrently the loop over the triples of the test set.

4.4. Results

Parameter learning was performed on the Leonardo HPC system of Cineca, using machines with one Intel Xeon Platinum 8358, 2.60GHz, 32 cores, 481GB of RAM, and Nvidia A100 GPUs with 64GB of memory. The Weighted Sampling algorithm was run on a machine running at 2.40 GHz with 16 GB of RAM with a time limit of 8 hours.

Table 3 shows, for each dataset, the maximum length of the body of the rules and the number of rules generated with the rule generation methods described in Section 3.1, Section 3.2, and Section 3.3, that here we call respectively Paths, AC1+AC2+C and Weighted Sampling respectively.

Table 4 shows the results of the experiments: for each dataset and algorithm, we report the metrics Hits@1, Hits@3, Hits@5, Hits@10, and MRR together with the time taken by parameter learning. The algorithm Paths+Conf means that we used the confidence as the parameters of the rules without performing EM. Note that the Weighted Sampling algorithm does not have an EM phase. That is, the reported values are the ones obtained only by sampling rules and by weighting them as described in Section 3.3. The confidence is computed using PyClause.

From Table 4 we can see that, for the rules found using the method described in Section 3.1 (called Paths), performing parameter learning using EM is not beneficial. This seems surprising, as tuning the parameters to improve the log-likelihood should produce a better classifier. It may be due to the fact that we had to subsample positive examples and to artificially generate negative examples, for which the test of absence from the training and validation set may not ensure absence from the test set or falsity in general. Also Weighted Sampling has limited performance, since it can only handle two of the four datasets. This is possibly due to the high number of instantiations that need to be found to associate weights with rules. Lastly, Table 4 shows that the approach described in Section 3.2, in which an input theory composed of **AC1**, **AC2**, and **C** rules whose parameters are then learned with LIFTCOVER+, could not be used. Specifically, we proved that although the approach works for small datasets, such as Nations, with very large datasets such as the other three considered, parameter learning with LIFTCOVER+ requires too much memory.

In order to compute the metrics for Paths, we used PyClause with the Noisy-Or aggregation strategy because it was significantly faster than our implementation in Prolog of Algorithm 1. This result is surprising as well, as Prolog systems such as SWI are supported by decades of software engineering expertise in improving the speed of answering this kind of queries, with advanced techniques such as argument indexing. This may be due to the fact that the queries are particularly simple and involve terms that are constants, for which the unification algorithm probably requires an excessive overhead because of its generality. Moreover, PyClause implements the optimizations discussed in Section 4.3 that, while leading to approximate results, could be the reason for the speed. Finally, another possible reason is that metrics computation in PyClause is implemented in C++.

5. Related Work

The KGC task has been studied for many years and a plethora of techniques has been developed to address them. A comprehensive survey can be found in [1]. Most of them are based on deep learning techniques. However, some notable exceptions exist, such as AnyBURL and SAFRAN, discussed in previous sections. Another approach can be found in [29] where the authors propose the adoption of so-called "soft" rules that are integrated within a probabilistic model. They address the KGC task using EM where the E step is solved via belief propagation while they develop an ad-hoc solution for the M step. With this methodology, they obtain competitive results w.r.t. other approaches such as AMIE [30] and TransH [31].

There is a large body of research also on parameter learning in StarAI literature and in particular on probabilistic logic languages. Most of them are based on EM, such as [10, 32, 33, 34], even if some exceptions exist [35, 36]. Here, we consider LIFTCOVER+ because it adopts the class of liftable PLP, where the inference process can be performed in a much faster way than in traditional PLP, due to the restricted types of clauses allowed. This, on the one hand, benefits the parameter learning process. However, the restriction on the types of rules may limit the expressiveness of the language and not capture complex relations. Further exploration of this aspect, possibly considering multiple layers of rules, even from a theoretical perspective, is an interesting future work.

6. Conclusions

In this paper, we leveraged Expectation Maximization algorithms for parameter learning in Probabilistic Logic Programming to solve the Knowledge Graph Completion task. We presented three different approaches to generate rules and learn their probabilities with LIFTCOVER+. Empirical results show that EM applied to liftable probabilistic logic programs to solve the KGC task seems to be not beneficial. Furthermore, the computation of the full rank is often unfeasible for datasets except for smaller ones. Future work includes approaching the KGC task with other PLP languages as well as studying alternative ways to generate rules.

Acknowledgments

This work has been partially supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - Next Generation EU (NGEU), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215. Damiano Azzolini, Elisabetta Gentili, and Fabrizio Riguzzi are members of the Gruppo Nazionale Calcolo Scientifico – Istituto Nazionale di Alta Matematica (GNCS-INdAM). We acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support. Elisabetta Gentili contributed to this paper while attending the PhD programme in Engineering Science at the University of Ferrara, Cycle XXXVIII, with the support of a scholarship financed by the Ministerial Decree no. 351 of 9th April 2022, based on the NRRP - funded by the European Union - NextGenerationEU - Mission 4 "Education and Research", Component 1 "Enhancement of the offer of educational services: from nurseries to universities" - Investment 4.1 "Extension of the number of research doctorates and innovative doctorates for public administration and cultural heritage.

References

[1] T. Shen, F. Zhang, J. Cheng, A comprehensive overview of knowledge graph completion, Knowledge-Based Systems 255 (2022) 109597. doi:10.1016/j.knosys.2022.109597.

[2] F. Riguzzi, Foundations of Probabilistic Logic Programming Languages, Semantics, Inference and Learning, Second Edition, River Publishers, Gistrup, Denmark, 2022.

[3] C. Meilicke, M. W. Chekol, D. Ruffinelli, H. Stuckenschmidt, Anytime bottom-up rule learning for knowledge graph completion., in: IJCAI, 2019, pp. 3137–3143.

[4] Z. Chen, Y. Wang, B. Zhao, J. Cheng, X. Zhao, Z. Duan, Knowledge graph completion: A review, IEEE Access 8 (2020) 192435–192456.

[5] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, Advances in neural information processing systems 26 (2013).

[6] C. Meilicke, M. W. Chekol, P. Betz, M. Fink, H. Stuckeschmidt, Anytime bottom-up rule learning for large-scale knowledge graph completion, The VLDB Journal 33 (2024) 131–161. doi:10.1007/s00778-023-00800-5.

[7] C. Meilicke, Personal communication, 2024.

[8] P. Betz, C. Meilicke, S. Ott, L. Galárraga, PyClause, 2024. URL: https://github.com/symbolic-kg/PyClause/.

[9] S. Ott, C. Meilicke, M. Samwald, Safran: An interpretable, rule-based link prediction method outperforming embedding models, arXiv preprint arXiv:2109.08002 (2021).

[10] T. Sato, A statistical learning method for logic programs with distribution semantics, in: L. Sterling (Ed.), Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995, MIT Press, 1995, pp. 715–729. doi:10.7551/mitpress/4298.003.0069.

[11] L. De Raedt, A. Kimmig, Probabilistic (logic) programming concepts, Machine Learning 100 (2015) 5–47. doi:10.1007/s10994-015-5494-z.

[12] F. Riguzzi, E. Lamma, M. Alberti, E. Bellodi, R. Zese, G. Cota, Probabilistic logic programming for natural language processing, in: F. Chesani, P. Mello, M. Milano (Eds.), Workshop on Deep

Understanding and Reasoning, URANIA 2016, volume 1802 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, 2017, pp. 30–37.

[13] A. Nguembang Fadja, F. Riguzzi, Probabilistic logic programming in action, in: A. Holzinger, R. Goebel, M. Ferri, V. Palade (Eds.), Towards Integrative Machine Learning and Knowledge Extraction, volume 10344 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 89–116. doi:10.1007/978-3-319-69775-8_5.

[14] J. Vennekens, S. Verbaeten, M. Bruynooghe, Logic Programs With Annotated Disjunctions, in: 20th International Conference on Logic Programming (ICLP 2004), volume 3132 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 431–445.

[15] A. Nguembang Fadja, F. Riguzzi, Lifted discriminative learning of probabilistic logic programs, Machine Learning 108 (2019) 1111–1135.

[16] D. Poole, First-order probabilistic inference, in: G. Gottlob, T. Walsh (Eds.), IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003, Morgan Kaufmann Publishers, 2003, pp. 985–991.

[17] A. P. Dempster, N. M. Laird, D. B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, Journal of the Royal Statistical Society: Series B 39 (1977) 1–38.

[18] D. Azzolini, E. Gentili, F. Riguzzi, Link Prediction in Knowledge Graphs with Probabilistic Logic Programming: Work in Progress, in: J. Arias, S. Batsakis, W. Faber, G. Gupta, F. Pacenza, E. Papadakis, L. Robaldo, K. Ruckschloss, E. Salazar, Z. G. Saribatur, I. Tachmazidis, F. Weitkamper, A. Wyner (Eds.), Proceedings of the International Conference on Logic Programming 2023 Workshops co-located with the 39th International Conference on Logic Programming (ICLP 2023), volume 3437 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 1–4.

[19] E. Gentili, Knowledge Graph Completion with Probabilistic Logic Programming, in: V. Poggioni, S. Rossi (Eds.), Proceedings of the AIxIA Doctoral Consortium 2023 co-located with the 22nd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2023), volume 3670 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023.

[20] A. Nguembang Fadja, F. Riguzzi, E. Lamma, Learning hierarchical probabilistic logic programs, Machine Learning 110 (2021) 1637–1693. doi:10.1007/s10994-021-06016-4.

[21] C. Andersen, T. Swift, The Janus System: A Bridge to New Prolog Applications, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, Springer Nature Switzerland, Cham, 2023, pp. 93–104. doi:10.1007/978-3-031-35254-6_8.

[22] T. Swift, C. Andersen, The Janus System: Multi-paradigm Programming in Prolog and Python, Electronic Proceedings in Theoretical Computer Science 385 (2023) 241–255. doi:10.4204/eptcs.385.24.

[23] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (2012) 67–96. doi:10.1017/S1471068411000494.

[24] T. Swift, D. S. Warren, Tabling with answer subsumption: Implementation, applications and performance, in: T. Janhunen, I. Niemelä (Eds.), Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings, volume 6341 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 300–312. doi:10.1007/978-3-642-15675-5_26.

[25] K. Toutanova, D. Chen, Observed versus latent features for knowledge base and text inference, in: A. Allauzen, E. Grefenstette, K. M. Hermann, H. Larochelle, S. W.-t. Yih (Eds.), Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality, Association for Computational Linguistics, Beijing, China, 2015, pp. 57–66. doi:10.18653/v1/W15-4007.

[26] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, in: C. Burges, L. Bottou, M. Welling, Z. Ghahramani, K. Weinberger (Eds.), Advances in Neural Information Processing Systems, volume 26, Curran Associates, Inc., 2013.

[27] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, T. Mitchell, Toward an architecture for never-ending language learning, in: Proceedings of the AAAI conference on artificial intelligence, volume 24, 2010, pp. 1306–1313.

[28] R. J. Rummel, Dimensionality of Nations Project: Attributes of Nations and Behavior of Nation

Dyads, 1950-1965, 1992. doi:10.3886/ICPSR05409.v1.

[29] R. Zhang, Y. Mao, W. Zhao, Knowledge graphs completion via probabilistic reasoning, Information Sciences 521 (2020) 144–159. doi:10.1016/j.ins.2020.02.016.

[30] L. A. Galárraga, C. Teflioudi, K. Hose, F. Suchanek, AMIE: association rule mining under incomplete evidence in ontological knowledge bases, in: Proceedings of the 22nd international conference on World Wide Web, 2013, pp. 413–422.

[31] Z. Wang, J. Zhang, J. Feng, Z. Chen, Knowledge graph embedding by translating on hyperplanes, in: Proceedings of the AAAI conference on artificial intelligence, volume 28, 2014.

[32] E. Bellodi, F. Riguzzi, Expectation maximization over binary decision diagrams for probabilistic logic programs, Intelligent Data Analysis 17 (2013) 343–363. doi:10.3233/IDA-130582.

[33] D. Fierens, G. Van den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory and Practice of Logic Programming 15 (2015) 358–401.

[34] D. Azzolini, E. Bellodi, F. Riguzzi, Learning the parameters of probabilistic answer set programs, in: S. H. Muggleton, A. Tamaddoni-Nezhad (Eds.), Inductive Logic Programming, Springer Nature Switzerland, Cham, 2024, pp. 1–14. doi:10.1007/978-3-031-55630-2_1.

[35] B. Gutmann, A. Kimmig, K. Kersting, L. D. Raedt, Parameter learning in probabilistic databases: A least squares approach, in: ECMLPKDD-2008, volume 5211 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 473–488. doi:10.1007/978-3-540-87479-9_49.

[36] D. Azzolini, F. Riguzzi, Optimizing probabilities in probabilistic logic programs, Theory and Practice of Logic Programming 21 (2021) 543–556. doi:10.1017/S1471068421000260.