Transfer Learning in Probabilistic Logic Models

Pouya G. Omran, Kewen Wang, and Zhe Wang

Griffith University, Australia,
pouya.ghiasnezhadomran@griffithuni.edu.au,
{k.wang, zhe.wang}@griffith.edu.au

Abstract. Several approaches to learning probabilistic logic programs have been proposed in the literature. However, most learning systems based on these approaches are not efficient for handling large practical problems (especially, in the case of structure learning). It has been a challenging issue to reduce the search space of candidate (probabilistic) logic programs. There is no exception for SLIPCOVER, a latest system for both parameter and structure learning of Logic Programs with Annotated Disjunction (LPADs). This paper presents a new algorithm T-LPAD for structure learning of LPADs by employing transfer learning. The new algorithm has been implemented and our experimental results show that T-LPAD outperforms SLIPCOVER (and SLIPCASE) for most benchmarks used in related systems.

Keywords: Transfer learning, probabilistic logic programs

1 Introduction

In logic programming, researchers have realized the importance of modeling uncertainty for a long time and thus various types of probabilistic logic programs have been proposed and studied. Consequently, significant attention has been paid to the issue of learning probabilistic logic programs. Recently, several approaches have been proposed for parameter learning (i.e., for a candidate logic program, the task is to determine probability degrees for certain objects such as rules and/or atoms). For instance, the PRISM [16] is well known for its distribution semantics. LeProbLog [7] is based on the technique of gradient descent while LFI-ProbLog [8] and EMBLEM [2] adopt an Expectation Maximization approach in which the expectations are computed directly using binary decision diagrams (BDDs). In many realistic applications, we need also to learn candidate logic programs as well as learning parameters. This paradigm of probabilistic logic program learning is referred to as structure learning. Obviously, this task is much more difficult than parameter learning only. ProbLog is proposed by De Raedt et al [5] for learning the structure of (probabilistic) logic programs while SEM-CP-logic is developed for learning ground LPAD programs. More recently, based on a new beam search, SLIPCASE [13] is developed for learning LPAD programs. SLIPCASE can learn general LPADs including non-ground programs. An improved version of SLIPCASE, named SLIPCOVER, is described in [1].
Transfer Learning with LPAD

learning algorithms for both SLIPCASE and SLIPCOVER, beam search is performed in the space of LPADs using the log likelihood of training data as the guiding heuristics and theory refinements are achieved using EMBLEM. However, there is still significant room for improving the efficiency of these algorithms. Especially, in these algorithms, the search space of LPADs is still very large and thus it would be useful to reduce the search space for structure learning algorithms using information from another application domain that shares a certain similarity with the domain of interest.

Human beings can make analogy across different domains by determining the structural similarities even in seemingly irrelevant domains, for example, we can easily understand the analogy between the domain of movie information and the domain of academic information. Even though the movie domain has nothing in common with the academic domain, we can still make the analogy based on certain similarity. For example, the predicate “movie(title,person)” with two arguments of types “title” and “person” is similar to the predicate “publication(person,title)” in the academic domain; the predicate “director(person)” is similar to “professor(person)”; the predicate “actor(person)” is similar to “student(person”).

In fact, reusing knowledge across different domains for learning has been actively pursued in machine learning communities [15, 4] and is usually referred to as transfer learning. It aims to learn a more accurate set of rules by using additional data from a source domain with less training data from target domain and less running time.

In this paper, we apply transfer learning in learning LPADs and describe a new algorithm T-LPAD for structure learning of LPADs (here ‘T’ in T-LPAD is for ‘Transfer learning’). Specifically, suppose that our task is to produce a set of LPAD rules for a domain of interest (i.e., target domain) based on a given training (relational) dataset. At the same time, we are given a set of LPAD rules in another domain (i.e., source domain) that is independent of the target domain but shares a kind of structural similarity with the target domain. By employing a technique from ontology matching [9], we are able to figure out a similarity degree for a pair of predicates in the source domain and the target domain. Using these similarity degrees, we construct rules in the target domain based on those rules in the source domain. This will allow us to narrow down the search space of constructing candidate rules in structure learning of LPADs. Note that these rules are standard (disjunctive) rules but not LPAD rules since no probability degrees are assigned to head atoms. For example, from the academic domain, we can produce a rule \( \text{director(person)} \lor \text{starring(person)} \leftarrow \text{movie(title,person)}, \text{bigname(person)} \). In the next stage, we figure out the best probability for each predicate in the rule head for each disjunctive rule using existing parameter learning algorithms such as EMBLEM. For instance, we can come up with an LPAD rule like \( \text{director(person)} : 0.4 \lor (\text{starring(person)}, 0.6) \leftarrow \text{movie(title,person)}, \text{bigname(person)} \).

We have developed a prototype implementation for T-LPAD algorithm and conducted experiments on four benchmarks IMDB, UW-CSE, WebKB, and
Twitter, which are widely used and publicly available [19, 1]. The experimental results show that T-LPAD outperforms SLIPCOVER (and SLIPCASE), two latest algorithms for structure learning of LPADs.

In the rest of the paper, we briefly recall basics of probabilistic logic programs and structure learning of such logic programs in Section 2, our new learning algorithm T-LPAD is described through a running example in Section 3, some experimental results are reported in Section 4, the related works are reviewed briefly in Section 5, and finally we conclude the work in Section 6.

2 Learning with Probabilistic Logic Programs

In this section, we review some basics of learning probabilistic logic programs, especially, structure learning of logic programs with annotated disjunctions (LPADs).

LPADs allow disjunction in the heads of program rules to express probabilistic multiple choices [20]. Similar to other classes of (probabilistic) logic programs, the fragment of LPD logic is based on three types of symbols: constants, variables, and predicates. Constants are names for specific objects in the domain, variables range over objects in the domain, and predicates represent relations or features among objects. We use $c$ (possibly with subscripts) or words starting with a lower case symbol for constants; $x$, $y$ and $z$ or with subscripts for variables; $P$ or words starting with a capital symbol for predicates. Variables and constants are typed. An atom is of the form $P(t_1, \ldots, t_n)$ where $P$ is a predicate and each $s_i$ is a constant or variable ($1 \leq i \leq n$). $P(t_1, \ldots, t_n)$ is a ground atom if every term $t_i$ is a constant. A probabilistic atom is a pair $(a, p)$ where $a$ is an atom and $p$ is a probability degree ($0 \leq p \leq 1$). $(a, p)$ is to represent that the atom $a$ is true with the probability $p$. A literal is an atom or its negation.

Formally, an LPAD rule is of the form

$$(a_1 : p_1); \cdots; (a_m : p_m) \leftarrow b_1, \ldots, b_n$$

where $(a_1 : p_1), \ldots, (a_m : p_m)$ are probabilistic atoms, $b_1, \ldots, b_n$ are literals. ':' is for disjunction. Informally, the above rule reads that if $b_1, \ldots, b_n$ are true, then at least one of $(a_1 : p_1), \ldots, (a_m : p_m)$ is true.

The semantics of LPADs is defined by the distribution semantics, which is first introduced for PRISM [16]. In the distribution semantics for LPADs, each ground is established by choosing one atom from the head of grounding form of an LPAD rule, and the probability $p_i$ of head atom $(a_i, p_i)$ is computed by accumulating the probability of worlds whose model evaluates $a_i$ as true. We refer the reader to [14] for further technical details.

Parameter learning for LPAD is carried out by the EMBLEM [2], which performs the Expectation Maximization (EM) over BDDs. A typical input for EMBLEM will be a set of ground atoms, a set of LPAD rules, and a set of so-called goal predicates. For each goal predicate, EMBLEM creates a BDD encoding its explanations and starts the EM cycle, in which the steps of Expectation and Maximization are continued until the log likelihood of the examples achieves a local maximum.
For structure learning of probabilistic logic programs, the learner needs to construct a logic program as well as determining relevant parameters. For instance, SLIPCOVER learns an LPAD program by first searching for promising ones from the space of disjunctive rules, looking for good refinements in terms of LL (likelihood) of the data, and finally performing EMBLEM on the best target disjunctive program. SLIPCOVER is able to learn general LPADs including non-grounded programs.

3 T-LPAD

In this section, we present a new approach to learning LPAD rules through transferring knowledge across domains. We will first provide a sketch of our learning algorithm T-LPAD and then explain further technical details in the subsections through a running example.

Suppose that we have two application domains in hand. One is the source domain, which is relatively well understood, and the other is the target domain in which we want to learn new knowledge in the form of LPAD rules. More formally, the source domain has a set of relational data and a set of LPAD rules that have previously obtained. The target domain has only a set of relational data but does not have any rules. The transfer learning task for the given domains is that, given a predicate in the target domain, we construct a set of LPAD rules for the target domain that are compatible with the (training) data in the target domain.

In our T-LPAD algorithm, for each domain we first construct a language bias called predicate description (PD), which contains information about predicate arity, predicate arguments, and their types. A PD can also be represented as a matrix (PD matrix). Based on two PD matrices for the source domain and the target domain, for each target predicate \( P \), we are able to determine a set \( S(P) \) of source predicates that are structurally similar to the given target predicate. In this way, we can construct a set of LPAD rules for the target domain from an LPAD rules in the source domain, by replacing each source predicate with a target predicate that is structurally similar to it.

In the next two subsections, we will explain our method using the following example.

Example 1 (Running Example). The source domain and the target domain are Academic and Movie, respectively. The academic covers knowledge about people in an academy department (i.e. students and professors) and their relationships. The movie domain contains information about movies, their directors, and the actors of the movies. The predicate descriptions for a Academic domain and the Movie domain are specified as follows.

**Academic PD:** advisedby(person, person), professor(person), student(person), publication(person, title), position(person, +pos)

**Movie PD:** workunder(person, person), actor(person), movie(title, person), director(person)
Academic domain contains one rule \( \text{professor}(x) : 0.2 \leftarrow \text{publication}(x,y) \).

Based on the above information, we want to learn rules for Movie about director (i.e., rules with director in the head).

### 3.1 PD Graph and Similarity Matrix

To extract similarity between the predicates in the source domain and those in the target domain, we make use of the PDs, which describes the predicates in the domain, their arities, the types of their arguments, and how each argument appears in rules (i.e., as variables or constants).

The predicate description for a domain can be conveniently represented as a graph called the PD graph for the domain.

**Definition 1** Given a domain \( D \), its PD graph, denoted \( G_D \), is defined as follows:

1. The vertex of \( G_D \) is either a predicate \( P/m \) with \( m \) denoting its arity, or a type \( T \) or \( +T \) with \( + \) denoting it is a constant type.
2. If the \( k \)-th argument of a predicate \( P \) is the type \( T \), then there is a directed edge labelled \( \#k \) from vertex \( P/m \) to vertex \( T \).

The PD graphs for those two domains in Example 1 are shown in Fig. 3.1.

![PD graphs for two domains UWC-SE and IMDB.](image)

Fig. 1. PD graphs for two domains UWC-SE and IMDB.

We want to construct a similarity matrix using PD graphs, in which each pair of source and target predicates is assigned a similarity degree as a real number. To achieve this, we adapt existing graph matching techniques which can extract similarity degree between the vertices of two graphs based on their structural similarity. Our PD graphs can be conveniently represented as RDF triples [9], where an RDF triple represents one edge or a labelled vertex of a PD graph.

Our RDF encoding contains three kinds of triples:
1. The predicate-type triples \((P, \#k,T)\) expressing that the \(k\)th argument of predicate \(P\) has the type \(T\). Each triple of this type encodes one edge in the PD graph.

2. The predicate triples \((P,n,predicate)\), encoding that vertex \(P\) represents a predicate with arity \(n\).

3. The type triples which has form \((T,cons/var,type)\), encoding that vertex \(T\) represents a constant or variable type.

Encoding PD graphs as RDF triples allows us to adapt the RDF matcher GMO \([9]\) to extract a graph matching matrix, denoted \(GM\), which can be seen as a function that maps each pair of predicates to a real number. We use \(GM\) as a basis to construct the similarity matrix between source and target predicates.

The graph matching matrix \(GM\) is completely based on structural similarity, which does not necessarily reflect the different labels in graphs, for example, whether a vertex is a binary predicate or a constant type. Hence, we need to refine \(GM\) to reflect the different reasons for two predicates to be considered similar, i.e., due to the same arity or similar argument types. We compute a label matrix \(LM\) to capture the similarity due to different labels, by comparing labels using an adapted string matching method \([9]\). We refine \(GM\) by compositing \(GM\) and \(LM\) linearly with parameter \(\beta\) into a refined matrix \(RM\) as the following.

\[
RM = \beta \cdot LM + (1 - \beta)GM
\]  

The refined matrix \(RM\) can be divided into two matrices, \(RM_t\) and \(RM_p\) which present the similarity between types and predicates respectively. To obtain the final similarity matrix \(SM\), we further refine \(RM_p\) through \(RM_t\) according to the argument types of predicates. For this refinement step, we use the linear combination (2). Again, each matrix can be seen as a function mapping each pair of predicates (or types) to a real number. For a predicate \(P\), \(T^P\) is the set of all the types of the arguments of \(P\), and \(T^P_i\) is the type of \(i\)th argument of \(P\).

To define \(SM\), for a pair of predicates \(P\) and \(Q\),

\[
SM(P,Q) = (1 - \alpha) \cdot RM_p(P,Q) + \alpha \cdot \frac{1}{m} \sum_{i,j} \max(RM_t(T^P_i, T^Q_j))
\]  

where \(m = \max(|T^P|, |T^Q|)\) and \(n = \min(|T^P|, |T^Q|)\). The parameter \(\alpha\) determines the degree how much the similarity degrees in \(RM_p\) are refine by the similarity between types in \(RM_t\). In particular, the second part of Equation (2) aggregates the similarities degrees of types for the two predicates \(P\) and \(Q\). For example, to compute the similarity degree between two predicates \(student(person)\) and \(movie(title,person)\), we have \(m = 2\) and \(n = 1\). Taking \(\alpha = 0.5\), suppose \(RM_p(student,movie) = 0.3\), \(RM_t(person,title) = 0.3\), and \(RM_t(person,person) = 1\), we can compute \(SM(student,movie) = 0.5 \times 0.3 + 0.5 \times 0.5 \times \sum_1 \max(0.3,1) = 0.4\).

We summarize the computation of the similarity matrix in Algorithm 1.

**Example 2 (Cont’d Example 1).** Taking as input the PDs of Academy and Movie domains, Algorithm 1 outputs the similarity matrix \(SM_{Academy,Movie}\) as follows:
Algorithm 1 Compute the similarity matrix

**Input**: source PD \( S \) and target PD \( T \)

**Output**: similarity matrix \( SM \)

1. Encode \( S \) and \( T \) into RDF documents
2. \( GM \leftarrow GMO^*(S,T) \) \( \triangleright \) \( GMO^* \) is our adapted GMO method
3. \( LM \leftarrow LabelStringMatch(S,T) \)
4. \( RM \leftarrow \text{Refine}(GM,LM) \) \( \triangleright \) \( \text{Refine} \) is defined in Equation (1)
5. \( SM \leftarrow \text{FurtherRefine}(RM_p,RM_t) \) \( \triangleright \) \( \text{FurtherRefine} \) is defined in Equation (2)

\[
\text{SM}_{\text{Academy,Movie}} = \begin{bmatrix}
\text{advised by} & 0.85 & 0.5 & 0.5 & 0.7 \\
\text{student} & 0.5 & 0.8 & 0.8 & 0.4 \\
\text{professor} & 0.5 & 0.8 & 0.8 & 0.4 \\
\text{position} & 0.7 & 0.4 & 0.4 & 0.8 \\
\text{publication} & 0.7 & 0.4 & 0.4 & 0.8
\end{bmatrix}
\]

3.2 Rule Construction

In what follows, we show how rules can be constructed for the target domain based on the similarity matrix. In contrast to existing approaches, which transfer second order template, we use rule templates of the form \( P_1; \cdots; P_m \leftarrow Q_1, \ldots, Q_n \) where \( P_i \) and \( Q_j \) are predicates (without arguments). We first show how our method works for rules with a single-atom head, and then extend the approach to rules with multiple-atom heads. For the convenience of discussion, we omit the weight of LPAD rules in our examples.

Let \( SM \) be the similarity matrix obtained previously, for each predicate \( P \) in the target (or source) domain, let \( S(P) \) (resp., \( T(P) \)) be the set of source (resp., target) predicates that have the highest similarity degree to \( P \) in \( SM \). To construct target rules with goal predicate \( G \), our rule construction method consists three steps.

In the first step, for each source predicate \( P \in S(G) \) and each rule \( r \) of the form \( P(t) \leftarrow Q_1(t_1), \ldots, Q_n(t_n) \) in the source domain, we obtain a set \( R_0(r) \) of rule templates in the target of the form \( G \leftarrow Q_1', \ldots, Q_n', \) where \( Q_i' \in T(Q_i) \). For example, suppose \( r \) is \( A(x,y) \leftarrow B(y), C(x,z) \), \( T(B) = \{ B', B'' \} \), and \( T(C) = \{ C' \} \). Then \( R_0(r) \) contains two rule templates: \( G \leftarrow B', C' \) and \( G \leftarrow B'', C' \).

In the second step, we assign variables and constants as arguments to the predicates in \( R_0(r) \). If a predicate has an argument type that is a constant type, we construct rules by assigning all possible constants of that type. Assignment of variable is less straightforward. We first assign a distinct variable to each argument which is not a constant, and then unify certain variables based on the variable sharing information in \( r \), which we capture using variable sharing constraints. A variable sharing constraint in \( r \) is of the form \( x(P_{m_1}, \ldots, P_{m_k}, Q_{n_1}, \ldots, Q_{n_l}) \), where \( P_{m_i} \)'s and \( Q_{n_i} \)'s are the head and body
predicates that have \( x \) as an argument. In the previous example, the variable sharing constraints in \( r \) are \( x(A,C) \) and \( y(A,B) \). Based on variable sharing constraints, we unify the assigned variable according to the following three conditions: (1) Two variable can unify only if they have the same type. (2) If two predicates share a variable in \( r \), the corresponding predicates in the rule template will have one variable unified whenever possible. (3) Unification is performed only when it is required by condition (1). In this way, we obtain a set of rules \( R_1(r) \) from \( R_0(r) \) through assignment. In the above example, suppose arities of \( G, B', B'' \) and \( C' \) are respectively 2, 1, 2 and 1, \( R_1(r) \) may contain some of the following rules (regardless of types):

\[
G(x,y) \leftarrow B'(x), C'(y), G(x,y) \leftarrow B'(y), C'(y), G(x,y) \leftarrow B''(x,z), C'(x), G(x,y) \leftarrow B''(y,z), C'(y), \\
and G(x,y) \leftarrow B''(z,x), C'(x).
\]

**Example 3 (Cont’d Example 2).** To learn a rule about director in the Movie domain, we obtain from our similarity matrix \( S(\text{director}) = \{\text{professor, student}\} \) and source rule \( r: \text{professor}(x) : 0.2 \leftarrow \text{publication}(x,y) \). Also, we have \( T(\text{publication}) = \{\text{movie}\} \). In this case we get one rule template in \( R_0(r) \), that is \( \text{director} \leftarrow \text{movie} \). After assigning arguments based on the single variable sharing constraint in \( r \), that is \( x(\text{professor}, \text{publication}) \), we obtain one rule in \( R_1(r) \), that is \( \text{director}(x) \leftarrow \text{movie}(y,x) \).

T-LPAD handles rules with multiple-atom head in a similar way. We illustrate this using an example. Suppose the source rule \( r \) is \( A(x,y) \leftarrow B(x,y) \leftarrow C(x), D(y) \) and from the similarity matrix we obtain \( T(A) = \{A', B'\}, T(B) = \{C'\}, T(C) = \{D'\}, \) and \( T(D) = \{E'\} \), then \( R_0(r) \) consists of the following two rule templates:

\[
A'; C' \leftarrow D', E' \text{ and } B'; C' \leftarrow D', E'.
\]

The assignment of argument works exactly as for rules with single-atom heads, and hence is not repeated here.

The rules constructed from the above two steps do not have weights in their head. Hence, in the third step, we generate weights for them. We initialise the weights of head atoms evenly, through dividing 1 by the number of head atoms. For example, in the cases with one head atom, noting that there is an implicit null atom, we assign 0.5 to each of these two. Then, we feed these rules to the parameter learner EMBLEM [2] to induce weights. Finally, rules with probability degrees below a threshold are eliminated from the candidates.

### 4 Experiments

In our experiments, we compare our T-LPAD with the state-of-the-art LPAD learner SLIPCOVER [1]. T-LPAD learns rules through transferring knowledge from an unrelated domain, whereas SLIPCOVER learns rules directly from facts. We try to address the following three questions:

- Can T-LPAD learn rules with high accuracy comparable to what SLIPCOVER can learn?
- Can T-LPAD learn such rules with (much) smaller amount of data than SLIPCOVER?
– Does T-LPAD learn such rules (much) faster than SLIPCOVER?

We used four domains IMDB, UW-CSE, WebKB, and Twitter, which are widely used and publicly available [19, 1]. Part of the IMDB domain about movies and the UW-CSE domain about academics have been used in the running example. The WebKB domain describes web pages from the computer science departments of four universities, and the Twitter domain contains tweets about Belgian soccer matches. As with existing approach, the facts in each domain are divided into smaller sets called folds [19, 1]. To evaluate the performance of learners on small amount of data, we further divide the existing folds into smaller ones. In our experiment, IMDB has 10 folds of facts, UW-CSE has 10 folds, WebKB has 8 folds, and Twitter has 8 folds.

To perform transfer learning, we paired up the domains into two pairs: IMDB with UW-CSE and WebKB with Twitter. The two domains in each pair serve as source and target respectively in one round of evaluation, and swap roles in another round. The goal predicate (i.e., that occurring in the head of rules to learn) for IMDB is workunder, for UW-CSE is advisedby, for WebKB is coursepage, and for Twitter is accountfan. These predicates were picked following the existing literature. Rules in each source domain either came with the domain or learnt from the facts.

For all learning tasks, we evaluated T-LPAD under the same configuration: graph matching steps ≤ 30 and convergence threshold $10^E - 9$, $\alpha = 0.5$, and $\beta = 0.3$. We used EMBLEM [2] to learn weights for the candidate rules. For SLIPCOVER, we adopted the parameters recommended by [1] for each domain. For a fair comparison, we disabled domain-dependent heuristics like “Lookahead” in SLIPCOVER. All the evaluation was conducted on a PC with 8 G RAM and corei5 CPU.

To evaluate and compare T-LPAD and SLIPCOVER, we adopted the standard measurement test set Log Likelihood (LL) and Area Under the Receiver Operating Characteristic Curve (AUROC) for measuring accuracy of the learnt rules [1]. LL directly measures the quality of the probability estimates produced, and the advantage of the AUROC is that it is insensitive to the large number of true negatives and it used both sensitivity and specificity for all possible thresholds.

<table>
<thead>
<tr>
<th>Number of folds</th>
<th>UW-CSE→IMDB</th>
<th>IMDB→UW-CSE</th>
<th>Twitter→WebKB</th>
<th>WebKB→Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>-891</td>
<td>-8301</td>
<td>-1429</td>
<td>-1316</td>
</tr>
<tr>
<td>SLIPCOVER</td>
<td>-590</td>
<td>-199</td>
<td>-533</td>
<td>-506</td>
</tr>
<tr>
<td>T-LPAD</td>
<td>-606</td>
<td>-601</td>
<td>-727</td>
<td>-907</td>
</tr>
<tr>
<td>AUROC</td>
<td>0.90</td>
<td>0.90</td>
<td>0.82</td>
<td>0.40</td>
</tr>
<tr>
<td>SLIPCOVER</td>
<td>0.88</td>
<td>0.91</td>
<td>0.87</td>
<td>0.48</td>
</tr>
<tr>
<td>T-LPAD</td>
<td>1.00</td>
<td>0.90</td>
<td>0.95</td>
<td>0.46</td>
</tr>
<tr>
<td>AUROC</td>
<td>0.90</td>
<td>0.90</td>
<td>0.85</td>
<td>0.37</td>
</tr>
<tr>
<td>SLIPCOVER</td>
<td>0.88</td>
<td>0.90</td>
<td>0.85</td>
<td>0.56</td>
</tr>
<tr>
<td>T-LPAD</td>
<td>0.99</td>
<td>0.93</td>
<td>0.89</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 1. Comparison between T-LPAD and SLIPCOVER.
Table 1 shows the LL and AUCROC for SLIPCOVER and T-LPAD with varying amounts of training data. In particular, “Number of folds” refer to the number of folds used for training the learners (T-LPAD uses training data only as inputs of EMBLEM), whereas the remaining data will be used for evaluating the learnt rules. The results show that regarding to the accuracy of learnt rules, T-LPAD was comparable to SLIPCOVER, and in several cases outperformed SLIPCOVER. The result was surprising as T-LPAD uses limited knowledge from an unrelated domain; yet on the other hand, it suggests the tight connection between the PD and rule structure of a domain. Through graph matching and the similarity matrix, T-LPAD is able to exploit such connection to a large extent and extract critical information for rule construction. The advantage of T-LPAD become obvious when the training data is reduced. In particular, with 1 fold of training data, SLIPCOVER suffered from insufficient training data whereas the performance of T-LPAD was reasonably stable. This is due to the fact that T-LPAD learns by transferring knowledge from another domain and hence is not data hungry.

We also measured the times needed for training with 6 folds, as shown in Table 2. All times are in minutes. From the results, it is clear that T-LPAD learns faster than SLIPCOVER when more training data are involved.

<table>
<thead>
<tr>
<th></th>
<th>SLIPCOVER</th>
<th>T-LPAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW-CSE→IMDB</td>
<td>1.1</td>
<td>0.1</td>
</tr>
<tr>
<td>IMDB→UW-CSE</td>
<td>37.63</td>
<td>0.1</td>
</tr>
<tr>
<td>Twitter→WebKB</td>
<td>5.25</td>
<td>3.15</td>
</tr>
<tr>
<td>WebKB→Twitter</td>
<td>2.1</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 2. Training times with 6 folds training data.

To conclude, with sufficient training data, the performance of T-LPAD is comparable to SLIPCOVER, while T-LPAD learns faster. In the case where only a small amount of training data is available, T-LPAD outperforms SLIPCOVER regarding to the quality of rules learnt.

5 Related Works

T-LPAD belongs to the class of deep transfer learning methods which are capable of generalizing knowledge cross distinct domains. Conceptually, the closest transfer learning approaches to T-LPAD are TAMAR [12], DTM [4], TODTLER [19], and transfer learning with type matching [11] which perform deep transfer in the context of Markov Logic Networks. One difference between our approach and these existing ones is about what knowledge is transferred. In particular, T-LPAD produces a similarity matrix, which associates the predicates of two domains based on their similarity in the predicate descriptions.

In [12, 19], the knowledge from source domain is transferred through an intermediate knowledge language called second-order templates. Recall our running
example, from the source rule $\text{professor}(x) : 0.2 \leftarrow \text{publication}(x, y)$, a second order template $X(x) \leftarrow Y(x, y)$ can be obtained. Assume we want to learn a rule about $\text{director}$ in the Movie domain, by initialising the second template, a desired rule $\text{director}(x) \leftarrow \text{movie}(y, x)$ cannot be obtained from the second order template. This is due to the fact that second order templates cannot capture the similarity between predicates beyond their arity and the order of their arguments. Yet from the PDs and especially the graphs in Figure 3.1, it is clear that predicate $\text{movie}$ in the Movie domain is most similar to the $\text{publication}$ in the Academy domain.

Existing approach [11] learns by transferring, instead of second order rule templates, type sharing knowledge among predicates. In particular, $\text{professor}$ and $\text{publication}$ share one type $\text{person}$ and one variable $x$ in the rule $\text{professor}(x) : 0.2 \leftarrow \text{publication}(x, y)$ of Academy. Rules $\text{director}(x) \leftarrow \text{workunder}(x, y)$ and $\text{director}(x) \leftarrow \text{actor}(x)$ can be constructed for Movie in [11] as candidates, as $\text{director}$ shares one type $\text{person}$ and one variable $x$ with $\text{workunder}$ in the former rule and $\text{actor}$ in the latter one. Yet again, these candidates are counter-intuitive. Matching types allows more flexibility than second order template (e.g., in the above example, $\text{actor}$ can be in the place of $\text{publication}$), but it does not take into account the similarity between predicates.

Another distinguishing feature of our approach is that our system does not require fact level data for structure learning. All the other approach require a refinement procedure to eliminate inaccurate rules, which is often data hungry. Also, the computational cost of such refinement is often high compared to the process of similarity matrix and rule construction in our case.

6 Conclusion

We have proposed an algorithm T-LPAD for LPAD structure learning using transfer learning, based on the similarity between two independent problem domains. Our algorithm identifies such similarity by matrix matching, and uses it to guide the candidate rule crafting for target domain in the presence of rules in the source domain. We have implemented the T-LPAD algorithm and conducted experiments for Web and social network domains. Our experimental results show that T-LPAD outperforms SLIPCOVER, a major structure learning algorithm for LPADs.

Acknowledgement This work was supported by Australian Research Council (ARC) under grant DP130102302.

References