Machine Learning (2020) 109:1435–1463
https://doi.org/10.1007/s10994-020-05871-x

Transfer learning by mapping and revising boosted relational dependency networks

Rodrigo Azevedo Santos[1] · **Aline Paes**[2] · **Gerson Zaverucha**[1]

Received: 15 February 2019 / Revised: 31 January 2020 / Accepted: 13 February 2020 /
Published online: 11 May 2020
© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2020

Abstract
Statistical machine learning algorithms usually assume the availability of data of considerable size to train the models. However, they would fail in addressing domains where data is difficult or expensive to obtain. Transfer learning has emerged to address this problem of learning from scarce data by relying on a model learned in a source domain where data is easy to obtain to be a starting point for the target domain. On the other hand, real-world data contains objects and their relations, usually gathered from noisy environments. Finding patterns through such uncertain relational data has been the focus of the Statistical Relational Learning (SRL) area. Thus, to address domains with *scarce, relational, and uncertain data*, in this paper, we propose TreeBoostler, an algorithm that transfers the SRL state-of-the-art Boosted Relational Dependency Networks learned in a source domain to the target domain. TreeBoostler first finds a mapping between pairs of predicates to accommodate the additive trees into the target vocabulary. After, it employs two theory revision operators devised to handle incorrect relational regression trees aiming at improving the performance of the mapped trees. In the experiments presented in this paper, TreeBoostler has successfully transferred knowledge between several distinct domains. Moreover, it performs comparably or better than learning from scratch methods in terms of accuracy and outperforms a transfer learning approach in terms of accuracy and runtime.

Keywords Transfer learning · Statistical relational learning · Theory revision

Editors: Dimitar Kazakov and Filip Železný.

✉ Aline Paes
alinepaes@ic.uff.br

Rodrigo Azevedo Santos
rodrigoasantos@cos.ufrj.br

Gerson Zaverucha
gerson@cos.ufrj.br

[1] Computing and Systems Engineering Program (PESC/COPPE), Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil

[2] Department of Computer Science, Universidade Federal Fluminense (UFF), Niterói, RJ, Brazil

1 Introduction

Machine learning algorithms have been widely and successfully used in many areas such as computer vision, robotics, social network analysis, among others (Lee et al. 2010; Sinapov and Stoytchev 2011). However, such success usually comes with the presence of large amounts of data. When the number of examples is relatively small, learning good models can be a challenging task. Lacking data is often the case for several real-world problems where collecting data is expensive or even impossible to obtain. To handle this issue, transfer learning techniques (Pan and Yang 2010) leverage a model learned from a source domain with more examples to learn from to be the start point of the learning process for another related domain where data is scarce.

Transfer learning has been widely employed in classical machine learning settings, such as for ensembles (Dai et al. 2007) and decision trees (Lee and Giraud-Carrier 2007). However, most of them do not take into account the relationships between entities of the domain and the fact that the examples may not be identically and independently distributed, which is the case for several real-world data. Learning from noisy relational data is the focus of the area called Statistical Relational Learning (SRL) (Getoor and Taskar 2007). Transfer Learning algorithms have also been developed in the context of SRL, mostly considering Markov Logic networks (MLN) (Richardson and Domingos 2006). Two of these algorithms (Davis and Domingos 2009; Van Haaren et al. 2015) transfer relational knowledge by creating a second-order representation of formulas from learned MLN. Other three algorithms (Mihalkova et al. 2007; Mihalkova and Mooney 2009; Kumaraswamy et al. 2015) find predicate mappings through searching methods to perform transference of clauses learned from MLNs by mapping their predicates.

Although these methods showed better results compared to learning MLNs from scratch, Natarajan et al. (2012) and Khot et al. (2011b) have shown that applying a boosted approach to learn SRL achieves the state-of-the-art in SRL tasks. Particularly, Relational Dependency Networks (RDN) yielded superior performance in terms of quality metrics and learning time over traditional SRL approaches. Based on the predicate mapping algorithm presented by Mihalkova et al. (2007) to transfer MLN clauses, we developed a similar predicate mapping approach to perform transference of Boosted RDNs, allowing for a transfer approached executed directly at the level of the additive models.

Thus, in this paper, we present a transfer learning algorithm called transfer Boosted RDNs by first mapping the predicates appearing in the trees. At a higher level, the algorithm generates the possible predicate mappings as it tries to transfer nodes from the source regression trees recursively. After finding such mappings, the algorithm propagates them to the rest of the trees. To complement the process and better adjust the mapped trees to the new target domain, TreeBoostler also includes a Theory Revision algorithm that proposes modifications to the mapped models in order to handle incorrectness and enhance the performance. Although in this paper we have focused on transferring Boosted RDNs, the proposed algorithm is general enough to be applied to other tree-based boosted SLR methods, such as Boosted MLNs.

We evaluated TreeBoostler in several real-world datasets and simulated the scenario where only a few data are available by training on one single fold and testing on the remaining folds. Furthermore, we simulate scenarios where only a minimal target data is

available (from one to 15 examples). Our results demonstrate that TreeBoostler can successfully transfer learned knowledge across different domains in a reduced runtime compared to another transfer learning algorithms. Also, transference is useful in terms of accuracy compared to learning from scratch methods based on (Boosted) RDNs and MLNs. An additional experiment was performed to investigate the behavior of the algorithm as the number of examples increases. The results demonstrate that TreeBoostler can be very competitive to traditional methods that learn from scratch even with the increase of the amount of data.

The remainder of the paper is organized as follows: the next section introduces the necessary background for one following our proposal. We review the algorithm for boosting RDNs, Theory Revision, and Transfer Learning. Next, we present the algorithm TreeBoostler and its mapping and revision components. Then, in the experimental section, we provide the results of the transfer learning algorithm applied in several datasets. Finally, we conclude with remarks and present possible directions for future research.

2 Background

In this section, we present a brief introduction about functional gradient boosting to learn relational dependency networks (RDNs) and the use of relational regression trees (RRTs) as functional gradients. Also, we give a brief introduction to Theory Revision and Transfer Learning.

2.1 Functional gradient boosting of relational dependency networks

Relational datasets can be represented by logical facts containing predicates and terms that are possibly augmented with argument types. In first-order logic (FOL), a predicate represents a relation between entities in the domain, such as a relation *published-by* that may connect the entities paper and person. Commonly, when representing relational datasets using FOL, the arguments may be associated with a type, for efficiency. The number of arguments a predicate takes is its arity. For example, the first argument of the predicate *published-by* exemplified earlier may be associated to the type *paper* and the second to type *person* (arity 2). An atom is a predicate applied to terms that can be variables, constants, or function symbols applied to terms. A literal is an atom or a negated atom and an atom whose terms are constants is called a ground atom (e.g., *published-by(paper1,person2)*). A definite clause is a disjunction of literals with exactly one positive literal.

In relational domains, the relations among objects are essential to allow the algorithms to discover rules and then to apply such rules to other objects to reach conclusions. Relational probabilistic models developed within the area of Statistical Relational Learning (SRL) (Getoor and Taskar 2007) extend this concept by combining the representational power of first-order logic with probability theory to handle uncertainty. Relational dependency networks (RDNs) (Neville and Jensen 2007) extends Dependency networks (DNs) (Heckerman et al. 2001), a probabilistic graphic model that allows cyclic dependencies, to relational domains and employs relational probability trees (RPTs) for the learning process.

RDNs approximate the joint distribution as a product of conditional probability distributions over ground atoms.

When learning RDNs, each conditional probability distribution can be represented as a relational probability tree (RPT)—as done by Neville et al. (2003)—in which leaf nodes report the number of positive and negative training examples that reached the leaf. Thus, a learning algorithm can induce a RPT for a given target predicate. Natarajan et al. (2012) proposed using relational regression trees (RRTs) instead of RPTs and presented an algorithm named RDN-Boost built upon functional-gradient boosting technique. The idea was to apply gradient boosting (Friedman 2000) to RDNs and represent each conditional probability distribution as a weighted sum of regression models. Explicitly, each relation is represented as a set of relational regression trees (Blockeel and De Raedt 1998). A particular advantage of the boosting method is that it allows learning both the structure and the parameters of RDNs simultaneously. In this paper, we refer to boosted trees as the set of these relational regression trees.

RDN-Boost uses RRTs and computes functional gradients for each training example. The functional gradient starts with an initial potential Ψ_0 and iteratively adds gradients Δ_i resulting after m iterations in the potential $\Psi_m = \Psi_0 + \Delta_1 + \cdots + \Delta_m$. After these m steps, the current model will have m regression trees for a given query predicate. Regression tree learner takes examples of the form $[(x_i, y_i), \Delta_m(y_i;x_i)]$ (weighted examples) and finds a regression tree h_m that minimizes $\sum_i [h_m(y_i;x_i) - \Delta_m(y_i;x_i)]^2$. The weight of an example corresponds to the gradient presented to that example. For each tree, the probability and the gradient of an example are computed based on its groundings. Then, the gradient serves as the weight for the example at the next training step.

In relational regression trees, inner nodes (or test nodes) are conjunctions of literals, and a variable presented in a node cannot appear in its right subtree (*i.e.*, variables are bounded along left-tree paths. This restriction is due to the fact that the right subtree is only relevant when the conjunction of literals fails.) (Gutmann and Kersting 2006). The algorithm learns a relational regression tree as follows: it starts with an empty tree and repeatedly searches for the best test for a node according to some splitting criterion. Then, it splits the examples in the node into *success* and *failure* according to the test. Examples covered by the clause reaches the left path (success), while examples not covered reaches the right path (failure).. The splitting criterion used was weighted variance on *success* and *failure*. For each split, the procedure is recursively applied further in order to obtain subtrees for the respective splits. The procedure stops if the variance in one node is small enough, the tree has reached a maximum depth defined in the procedure or has derived a maximum number of leaves. In the leaves, the average regression value is computed (Gutmann and Kersting 2006; Natarajan et al. 2012). An example is presented in Fig. 1. This tree was learned for the query predicate *advisedby* aiming at predicting if B advises A. In the tree, if A is a *student*, B is a *professor* and both work in the same publication (*publication*(C, B), *publication*(C, A), then the regression value is 0.858. On the other hand, if the node (*student*(A), *professor*(B)) is not satisfied, then the regression value is -0.142. Negative values indicate low probabilities and, for this tree, the fact that A is not a student or B is not a professor indicate a low probability of A to be advised by B. This regression tree learner also considers aggregation functions such as *count*, *max*, *average* in the inner nodes, however we did not consider aggregation functions in our work. For more details about aggregation functions, we refer the reader to Natarajan et al. (2012).

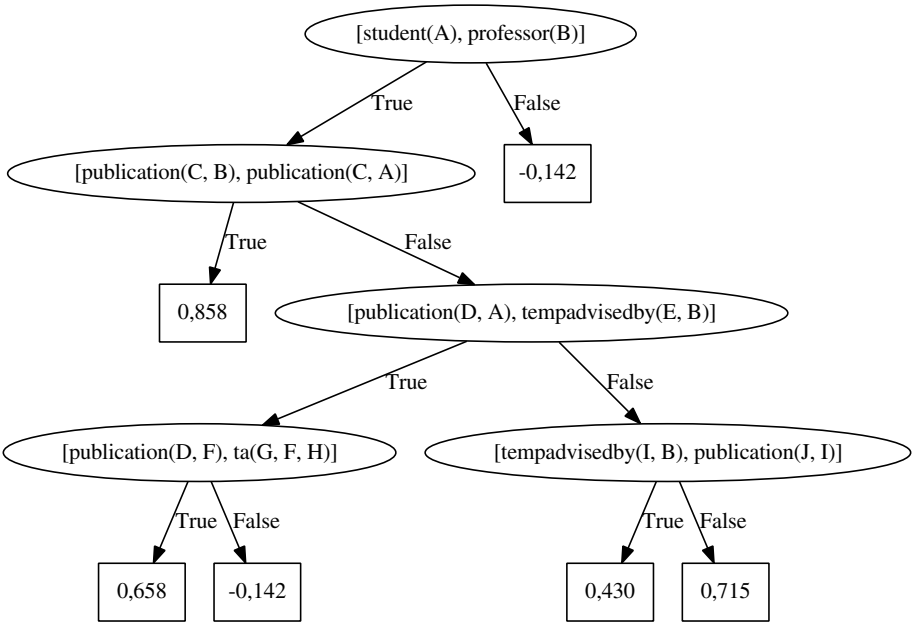

Fig. 1 An example of relational regression tree. The leaves are the regression values learned and nodes are conjunctions of literals

Note that each path between the root and leaf in the regression tree is as a clause in a logic program. For the Fig. 1, the left-most path in the tree is the clause: $student(A) \wedge professor(B) \wedge publication(C, B) \wedge publication(C, A) \Rightarrow advisedby(A, B)$. The algorithm evaluates the clause in order from left to right for a particular query. After that, it returns the corresponding regression value. In the logical setting, multiples ground clauses can be satisfied for a particular query. In the case of relational regression trees, this is avoided by considering only the first satisfied ground clause (*i.e.*, the first path that covered an example). This is equivalent to add a cut to the end of each clause in a logic program. Then, the second clause to be evaluated in the example if the first clause were not satisfied would be the clause made by the second left-most path.

2.2 Theory revision

Learning First-Order Logic theories from a set of examples and background knowledge (BK) is a process known as Inductive Logic Programming (ILP). Given the BK and a set of examples represented as logical facts, an ILP learner derives a hypothesis in the form of a logic program to cover as many as possible positive examples while avoiding covering negative examples. Because such systems start from an empty initial hypothesis, we say that they learn from scratch. However, an incomplete or only partially correct theory may exist, and one may take advantage of it as a starting point to improve it instead of discarding it and learning a new theory from scratch. An incomplete/partially correct theory may exist

because of different reasons: (1) it has been proposed by a domain expert, who has useful, but only partial knowledge about the domain; (2) it has become incomplete due to new data available; (3) it has come from a different yet related domain through transference. In such cases, the theory may contain important information that should not be discarded. Because of that, it is desirable to use the given theory as a starting point in the learning process. These crucial considerations have contributed to the development of Theory Revision systems (Richards and Mooney 1995; Duboc et al. 2009; Paes et al. 2017).

Theory revision is the process of repairing incorrect or incomplete theories from a given set of examples. This process is a sub-task of a general problem—improving the quality of a given theory—known as theory refinement. A theory revision task consists of proposing modifications to the theory which are going to imply in changes in the set of answers, *i.e.*, covering missing answers or fixing incorrect answers made by the theory. On the other hand, the theory restructuring task does not change the set of answers given by a theory. Theory revision is then defined as follows (Wrobel 1996): given an initial theory H and a set of positive and negative examples E^+ and E^- composing the set of examples E, the system aims at finding a revised theory H' that covers all the positive examples (completeness) and none of the negative examples (consistency), and also obeys a minimality criteria which requires minimal revisions of the theory. However, it is not always possible to find a complete and consistent theory; thus theory revision systems find theories as close as possible to be complete and consistent.

Usually, theory revision is applied when new data have become available or when we want to improve an imprecise theory learned from scratch. The initial theory is assumed to be partially correct, and thus, only some points are responsible for misclassifications in the dataset. Therefore, a theory revision system needs only to propose modifications to such points instead of discarding the initial theory or proposing modifications for all its clauses (Paes et al. 2017). These points are called revision points and are detected according to misclassified examples. When positive examples are not covered (*i.e.*, false negatives), the theory is too specific. Similarly, when negative examples are covered (*i.e.*, false positives), the theory is too general.

Revision operators are responsible for proposing modifications at each revision point, and the type of the revision point determines which revision operator to apply. Commonly, two types of revision operators are considered: (1) generalization operators; and (2) specialization operators (Wrobel 1996). Generalization operators can be used to handle false negatives, while specialization operators can be used to remove false positives. We refer the reader to Wrobel (1996) for more information about revision operators.

In this work, we refer to a theory as the boosted trees learned from a specific query predicate. Also, proposing modifications to clauses is equivalent to proposing modifications to paths in a tree. We will explain further the operators proposed for revising boosted trees in the Sect. 3.

2.3 Transfer learning

Traditional machine learning algorithms work with the assumption that both training and future data are in the same feature space and have the same distribution. However, this assumption may not hold in real-world scenarios. Then, when the test distribution changes

w.r.t. to the training data, the algorithms need to relearn the model from scratch using the newly collected data. Arguably, a more efficient solution would be to *adapt* the previously learned model to the new distribution of examples. Another situation that may benefit from adaptability is when one has a domain for which collecting data is quite expensive or even impossible. However, it could be the case that, while we do not have sufficient data for a specific domain, we may have plenty of data for a similar domain. Consider, as an example, learning from simulations. While obtaining real-world measures would be extremely expensive, generating data from simulations which have a different distribution comparing to the reality could be more accessible (*e.g.*, a physics engine mimicking movements of a robot).

Transfer learning (Pan and Yang 2010) addresses the problem of lacking data by allowing that domains used in training and testing be different. The advantage is to exploit the knowledge learned in a source domain to improve the performance of a related target domain. An example might be using the knowledge obtained to recognize a specific kind of object to help to recognize a similar object, or equivalently, using knowledge learned from Spanish and apply it to learn Portuguese. Indeed, Transfer learning is motivated by the fact that humans can take the knowledge learned in a specific domain and apply it to a completely different domain. The difference between traditional learning and transfer learning is that traditional learning tries to learn a task from scratch regarding one specific domain, while transfer learning tries to transfer knowledge learned from a previous source task to a target task.

A definition of Transfer learning was presented by Pan and Yang (2010) as the following: a domain $\mathcal{D}$ is consisted by a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$ where $X = \{x_1, \ldots, x_n\} \in \mathcal{X}$. Considering the problem of document classification as an example, then $\mathcal{X}$ is the space of all term vectors, x_i is the i-th term of a vector of a given document and X is a particular sample. Given a domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ consists of a label space $\mathcal{Y}$ and a not observed conditional probability distribution $P(Y|X)$ which could be learned from training data. In the document classification example, $\mathcal{Y}$ is the set of all labels. Finally, given a source domain $\mathcal{D}_S$ and a source task $\mathcal{T}_S$, as well a target domain $\mathcal{D}_T$ and a target task $\mathcal{T}_T$, the purpose of transfer learning is to help to learn the target conditional probability distribution $P(Y_T|X_T)$ in $\mathcal{D}_T$ using the knowledge obtained from $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$. If two domains are different, they have either a different feature space or different marginal probability distributions due to the definition of the domain as a pair $\mathcal{D} = \{\mathcal{X}, P(X)\}$. Thus, the condition $\mathcal{D}_S \neq \mathcal{D}_T$ implies that $\mathcal{X}_S \neq \mathcal{X}_T$ or $\mathcal{P}_S(X) \neq \mathcal{P}_T(X)$. In document classification, a different feature space may correspond to domains of two different languages and different probability distribution may correspond to domains in the same language but about different topics. Similarly, for a definition of task as a pair $\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$, the condition $\mathcal{T}_S \neq \mathcal{T}_T$ implies that $\mathcal{Y}_S \neq \mathcal{Y}_T$ or $P(Y_S|X_S) \neq P(Y_T|X_T)$.

Most of the current work conducted in Transfer Learning assumes that the source and target domains are related, *i.e.*, there exists some relationship between both feature spaces. In the relational data scenario, it is assumed that the source and target data may share similar relationships. If two domains are related to each other, there may exist two similar relationships that connect entities in a domain, and thus a mapping for these relationships may be found. For example, the role of a professor in a university domain

can be seen as similar to a director in a cinematographic domain because they play similar roles by teaching/leading students and actors. Also, the relationship between a professor and a student, as well as between a director and an actor, can be considered similar. Consequently, other relationships as "professor publishing a paper" and "a director directing a film" can also be considered similar. More examples would be the relationship of a professor teaching students which is similar to a football coach teaching football players, also a team playing a specific sport, which is similar to a company belonging to a specific economic sector.

There are three main essential research issues regarding transfer learning (Pan and Yang 2010). One issue is to decide what kind of knowledge to transfer between domains or tasks. Some knowledge may be shared between both the domains so that it could help improving performance in a target domain. For relational domains, the knowledge to be transferred is the structure of the theory and a mapping from source predicates to target predicates must be found in order to find which clauses to transfer to the target domain. In this work, we consider transferring the structure of each tree learned from the source domain by finding a mapping between the predicates in the trees. Second, how to perform transfer needs to be considered, and learning algorithms must be developed to accomplish this process. Most of the previous works have focused on these two issues. Thus, transfer algorithms consider what to transfer across domains and how to proceed with the transference. The third issue that needs to be solved is when to perform transfer which corresponds to answering when transferring should be done or not. In some situations, a *negative transfer* may hurt the learning performance in the target domain resulting in worse accuracy than to not transfer at all. Knowing when not to conduct the transfer is also an interesting issue in order to avoid struggling in a transfer that would lead to a worse result than to no transfer at all.

Some transfer learning methods in the SRL context were proposed before. The TAMAR algorithm (Mihalkova et al. 2007), for example, maps predicates in the clauses of an MLN learned from a source domain in order to transfer these clauses to a target domain. The legal mapping that gives the best-weighted pseudo-log-likelihood (WPLL) in the target domain is the mapping used for that clause. In a second step, TAMAR performs theory revision for the mapped structure through an algorithm similar to the FORTE algorithm (Richards and Mooney 1995) to improve its accuracy. Another example is the algorithm SR2LR (Mihalkova and Mooney 2009), an extension of TAMAR to deal with minimal target data. It considers the extreme case described as single-entity-centered where one entity is available in the target domain, although they are also generalized for more than one entity.

Another algorithm, DTM (Davis and Domingos 2009), uses second-order Markov Logic where formulas contain predicate variables. The key idea is to discover second-order structure shared by source and target domains by instantiating second-order formulas with predicates from the target domain. TODTLER algorithm (Van Haaren et al. 2015) also creates a second-order representation. It uses previous useful second-order patterns learned in the source domain to bias the learning process in the target domain towards models that also have these patterns. LTL (Kumaraswamy et al. 2015) compares types between source and target predicates and performs a matching. After that, it builds the first-order logic clauses in the target domain by performing a type-based tree construction. LTL algorithm

also performs theory refinement in its rules. Differently, LAST (Odom et al. 2016) algorithm allows for human expert advice in the process of refinement through the LTL algorithm. The goal is to improve the human-machine interaction by allowing the expert to refine a transferred knowledge.

Finally, (Ramon et al. 2007) also includes operators acting over trees; however, different from us, they focus on relational trees targeting reinforcement learning policies. They include an operator that replaces an internal sub-tree by another. We found out that implementing such an operator in boosted RDNs would bring additional complexity to our revision algorithm as we would have to carefully control the variables shared among the different nodes involved in the replacement. The behavior of replacing an internal sub-tree could be achieved by removing a sub-tree up to the leaf and adding another one in its place, with a successive application of our both revision operators, as we discuss later.

Our algorithm differs from the algorithms presented as we map and revise boosted models directly at the level of the additive models. Moreover, we built our implementation focusing on RDNs, instead of relying on MLNs, as most of the previous work. Our approach considers the set of relational regression trees learned from the source domain to bias the learning algorithm in order to obtain a target model derived from the source model. The algorithm starts by transferring the source tree structures and inner nodes to the target domain. This component is similar to TAMAR's predicate mapping algorithm. The next step is the process of theory revision where we contribute with finding revision points *and* applying two revision operators over relational regression trees. Proposing modifications in a tree affects the covering of examples in both its left and its right path. We explain the functioning of the TreeBoostler, our proposed algorithm, in the next section.

3 The TreeBoostler algorithm

The algorithm we devised in this work follows two top-level components: first, it transfers the source boosted trees structure to the target domain by finding an adequate predicate mapping; second, it revises those trees by pruning and expanding nodes. The regression values are learned simultaneously in both steps. Next, we detail each one of these steps.

3.1 Transferring the structure

A fundamental problem when tackling transfer learning on relational domains is to automatically find how to map the source vocabulary to the target domain. In this way, the first step of the overall process is to find this mapping, where we reduce the overall vocabulary of both domains to their set of predicates, making our first problem to find the best mapping of source predicates to target predicates. With that, the boosted trees learned from the source domain are transferred sequentially to the target domain, and their parameters are relearned to fit the target data. Mihalkova et al. (2007) introduced two approaches for establishing a predicate mapping regarding Markov Logic Networks: (1) a global mapping, which finds a corresponding target predicate to each source predicate and applies

this mapping to the entire source structure (i.e. all clauses) at once; and (2) a local mapping, which finds an independent predicate mapping for each independent part of the entire structure (i.e. each clause). This latter case constructs a predicate mapping only for the predicates that appear in a specific clause, separately, independently of how the predicates appearing in the other clauses have been mapped before. Generally, the local mapping approach is more scalable since the number of predicates that appears in a clause is naturally smaller than the total number of predicates of a source domain and more flexible, as the mapping in one part of the structure does not necessarily hold or depends on all the other rest of the structure.

In this work, we choose to follow the local approach by finding the best local predicate mapping for transferring the boosted trees. As we have mentioned earlier, each path from the root to a leaf in the relational regression tree can be seen as a clause in a logic program. However, these paths are not independent of each other as they may share the same inner nodes with different paths in the relational regression tree. Also, trees cannot be interpreted individually since each one depends on the previously handled trees. Thus, the algorithm translates the predicates presented in the inner nodes according to the previously found translations in order to keep the same mapped predicates along the entire process.

To find the best predicate mapping for the entire structure, we perform an exhaustive search through the space of all legal mappings of the predicates that are in the inner node which have not a translation yet. The legal mapping that provides to the node the best split is selected as the best node and mapped predicate. We defined the weighted variance as the split criterion. Transference starts from the root node of the first source tree and proceeds to find not-mapped predicates recursively (similar to learning from scratch) in order to update the current predicate mapping.

Definition 1 Let $p(X_1, \ldots, X_n)$ be an atom in the source vocabulary with predicate p and arity n. Let $q(Z_1, \ldots, Z_m)$ be an atom in the target domain with predicate q and arity m. Let $S = \{type_{s_1} \rightarrow type_{t_1} \ldots type_{s_n} \rightarrow type_{t_m}\}$ be the set of constrained types, where the first term of each element is a type in the source domain and the second term is a type in the target domain. We say that $p/n \rightarrow q/m$ is a legal mapping when $n = m$ (they have the same arity), and for each pair of corresponding terms (X_i, Z_i) where X_i is a term in $p(X_1, \ldots, X_n)$ and Z_i is a term in $q(Z_1, \ldots, Z_m)$, if X_i is associated to the type $type_{s_i}$ and Z_i is associated to the type $type_{t_i}$, then either $type_{t_i}$ has not appeared before as the second term of an element in S or $type_{s_i} \rightarrow type_{t_i} \in S$. The set of compatible types starts empty and is iteratively filled in with a type correspondence yielded from a predicate mapping.

We define a mapping as legal if each given source predicate is mapped to a compatible target predicate or an "empty" predicate. If the source and target predicates have the same arity and their argument types agree with the current type constraints they are considered compatible. The mapping is done by following the current type constraints which each type mapped to at most one corresponding type in the target domain. For example, the current type constraints are empty and the first predicate to map is *genre(person,genre)*, then the target domain predicate *projectmember(project,person)* is considered to be compatible. Therefore, the type constraints are updated with the following constraints: *person* → *project* and *genre* → *person*. Since all following predicates to be mapped need to conform to the current type constraints, a mapping for the predicate *advisedby(person,person)* can only be compatible with *sameproject(project,project)*. Algorithm 1 finds legal mappings

given the source predicates to be mapped, possible target predicates to consider and current predicate mappings and type constraints.

Algorithm 1 Finding legal mappings given the source and target predicates

```
function LEGAL_MAPPINGS(srcPreds, tarPreds, predsMapping, typeConstraints)
    mappingsList ← []
    Pick the first unmapped source predicate srcPred
    for each tarPred ∈ tarPreds do
        if isCompatible(srcPred, tarPred) then
            Add this mapping to a copy of predsMapping
            Update a copy of typeConstraints
            Call LEGAL_MAPPINGS with new parameters
            Insert mappings to mappingsList
        end if
    end for
    return mappingsList
end function
```

Note that the boosted trees are learned concerning a query atom; because of that, the transfer algorithm must receive as input the source and target query atoms to start the

Table 1 Predicate mapping automatically discovered for transferring IMDB→UW-CSE

workedunder(A,B)	→ advisedby(A,B)
director(A)	→ professor(A)
actor(A)	→ student(A)
movie(A,B)	→ publication(A,B)

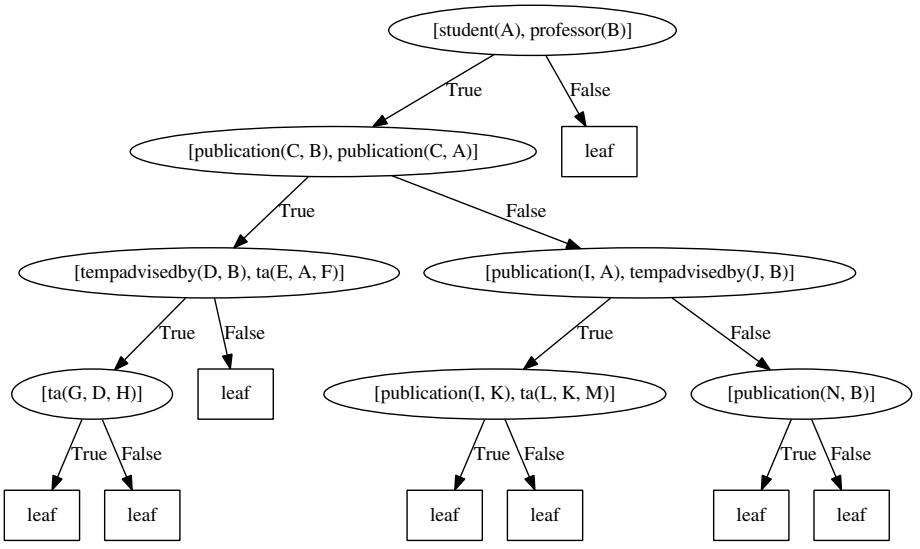

Fig. 2 One regression tree to be transferred from UW-CSE to Cora for query predicate *advisedby*. Regression values are not considered for transference. They are relearned in the process

transference. Hence, the predicate mapping starts with a mapping from the source query predicate to the target query predicate. For example, considering to transfer the source query atom *workedunder(person,person)* from IMDB dataset to the target query atom *advisedby(person,person)* from UW-CSE dataset, where *person* is the type of both arguments, in both target query domains. The algorithm starts the type constraints set with the mapping *person* → *person* and the predicate mapping set with *workedunder(A,B)* → *advisedby(A,B)*. Table 1 shows the final predicate mapping set, found after transferring the entire boosted tree structure.

In case the algorithm does not find a compatible mapping, a predicate in the source domain is mapped to an "empty" predicate. This is used to decide how to map the nodes in the trees, encompassing three cases: (1) all the literals in an inner node have a non-empty predicate mapping. This is the best scenario, as we can keep the same number of literals in the transferred tree; (2) an inner node has some predicate mapped to an "empty" one, but there is at least one predicate mapped to a non-empty, then the ones mapped to empty are discarded and the others remain; (3) an inner node has all their literals mapped to an empty predicate. This last case is the more complicated scenario, as discarding all the literals yields an empty node, which affects the tree structure, leading to no structure transference in the worst case. For example, the transference UW-CSE → Cora would result in a null theory as shown in Fig. 2 since Cora dataset has no unary predicates and the root nodes of learned source trees are conjunctions of unary predicates. To tackle the presented scenarios, the algorithm discards the "empty" node, promotes its left child and appends its right child to the right-most path of the subtree. If the left child is a leaf, then the "empty" node is discarded, and the right child is promoted. It is essential to mention that the transfer process is also subject to the search bias growing tree parameters, namely the maximum depth and the maximum number of leaves per tree. Generally, in our experiments, we restricted the size of the trees with the same parameters used for learning from scratch in the source domain. It means that the nodes and the subtrees appended to the right-most path of the tree may be ignored in the process. In some cases, the transference may result in inner nodes that cover all the examples in their left or right path, making the node with no examples useless. To reduce the tree depth, the algorithm discards such nodes and promotes the child that covers all examples. The Algorithm 2 presents the transfer mechanism described.

Our method includes three search bias to conduct the way the algorithm performs the mapping. The first one, called here as *searchArgPermutation*, allows searching for the permutation of all arguments in the target predicate to check if one of them makes the source and target predicates compatible. It allows for example, the mapping of a source predicate with the inverse relation of a target predicate (e.g. *workedunder(A,B)* → *advises(B,A)*, which is the same as *advisedby(A,B)*). The second search bias, named *searchEmpty*, allows generating an additional "empty" mapping even if there is a compatible target predicate to map the source predicate. The last one, named *allowSameTargetMap*, allows mapping distinct source predicates to the same target predicate. If this bias is not used, the algorithm finds a one-to-one correspondence between source and target predicates (except for "empty" mappings).

Algorithm 2 Top-Level Transfer Algorithm

Require: *theory*, a set of regression trees
Ensure: *transferred*, the transferred regression trees
  function TRANSFER(*theory*)
    transferred ← ∅
    for each *tree* ∈ *theory* **do**
      new_tree ← ∅
      TRANSFER_TREE(*tree.root*, *new_tree*)
      Append *new_tree* to *transferred*
    end for
    return *transferred*
  end function
  function TRANSFER_TREE(*node.root*, *transfer_node*)
    if *node* is leaf **then**
      Define *transfer_node* as leaf
      Stop procedure
    end if
    predicates ← Get set of predicates not mapped from *node*
    if *predicates* is empty **then**
      new_node ← Translates predicates in *node*
      transfer_node ← *new_node*
    else
      Call LEGAL_MAPPINGS given *predicates* and current predicate mappings and type constraints
      Generate possible nodes by translating predicates in *node* according to legal mappings
      Find the node that gives the best split
      Update the global variable *predsMapping* and *typeConstraints*
      transfer_node ← best node
    end if
    if *transfer_node* is not empty **then**
      Call TRANSFER_TREE(*node.left*, *transfer_node.left*)
      Call TRANSFER_TREE(*node.right*, *transfer_node.right*)
    else
      if *node.left* is leaf **then**
        Call TRANSFER_TREE(*node.right*, *transfer_node*)
      else
        Append *node.right* to to the right-most path of *node.left*
        Call TRANSFER_TREE(*node.left*, *transfer_node*)
      end if
    end if
  end function

3.2 Revising the structure

When transferring learned theories from one domain to another, it is usually not enough to map the vocabularies from both domains to achieve a model representative of the target domain (Mihalkova et al. 2007). Such theories may contain multiple faults that prevent them from correctly predicting examples due to the difference in the distribution of both domains. These faults can be repaired through the process of Theory Revision (Wrobel 1996). The main idea of Theory Revision is to search for points in the theory that are preventing the examples from being correctly classified and propose modifications to them. In a Transfer Learning scenario, the revision process attempts to adjust the initial mapped source theory to fit the target data. The goal is to achieve more accurate theories since the theory revision allows the learning algorithm to build clauses from partial or incomplete theories that would otherwise not be found in the constrained search space.

Our theory revision component follows the three major steps:

1. Searching for paths in the trees responsible for bad predictions of examples and defining them as revision points.
2. Proposing possible modifications to the revision points by applying the revision operators.
3. Scoring both transferred and revised theory and choosing to stay with the best one.

In the traditional Theory Revision literature concerning Inductive Logic Programming (ILP), the points to be changed are defined according to a misclassified example defined according to the proved examples, as explained in the Sect. 2.2. However, this concept does not hold for the Statistical Relational Learning (SRL) case, which considers the uncertainty of the domain. Thus, we define the points to be changed according to the bad predictions made by the trees. Here, a node is marked as "badly" predicting when its weighted variance is greater than a given threshold δ, reflecting the fact that a node is not good enough to stop the growth of its subtree.

Definition 2 *Revision Point* Let v be a leaf node in a tree and let δ_v be the weighted variance of examples being covered until v. Given a threshold δ, we say that v is "badly" predicting the examples when $\delta_v > \delta$. Hence, the leaf node v is marked as a *Revision Point*.

The revision points need to be modified during the revision process in order to increase accuracy. In the traditional ILP setting, examples incorrectly covered determine the revision operator to be applied: a positive example not covered by the theory indicates that the theory is too specific and needs to be generalized, on the other hand, a negative example covered by the theory indicates that the theory is too general and needs to be specialized. In the case of relational regression trees, positive and negative examples are covered by the paths in the tree, with their respective weights determining the weighted variance of the covered examples. In this way, instead of determining the type of the revision point (specialization or generalization), we only assume that some paths are responsible for harming the accuracy. To make this matter simpler, we define as a revision point any leaf that has a "bad" weighted variance as defined before. Arguably, modifications on the paths ending up on such leaves will change the way an example is covered, resulting in a differently weighted variance.

We designed two types of revision operators: (1) a pruning operator, which increases the coverage of examples by deleting nodes from a tree (and in such a way, it may be seen as a generalization operator); and (2) an expansion operator, which decreases the coverage of examples by expanding nodes in each tree (in the same way, it can be seen as a specialization operator). We describe them as follows:

- **Pruning** operator prunes the tree from the bottom to top by removing a node whose children are leaves marked as revision points.
- **Expansion** operator recursively adds nodes that give the best split in a leaf considered as a revision point.

The top-level theory revision algorithm fully applies the pruning and expansion operators in all the revision points at once. The first step is to call the Pruning procedure

for each tree in the model. The Pruning procedure receives a root node of a given tree as input and recursively removes nodes that contain leaves marked as revision points. However, this process may completely prune an entire tree, eventually leading to the deletion of all the trees. If this happens, the revision algorithm will face the expansion of nodes from an empty tree which is the same as learning from scratch. To avoid that, if the pruning results in a null model, the effect of this operator is ignored as if it was never applied.

Next, for each tree, the Expansion procedure is called, and recursively expands the revision points. The last step is done by scoring both the transferred theory (before applying theory revision) and the revised theory. The revised theory is implemented if it has a scoring better than before. The scoring function is the conditional log-likelihood (CLL) over the examples. The Algorithm 3 presents the theory revision process after mapping the vocabulary of the source and target domain.

Algorithm 3 Top-Level Theory Revision Algorithm

```
Require: theory, a set of regression trees
Ensure: new_theory, the possibly revised trees
  function REVISION(theory)
      new_theory ← ∅
      for each tree ∈ theory do
          new_tree ← PRUNING(tree)
          Append new_tree to new_theory
      end for
      if new_theory is null then
          new_theory ← theory
      end if
      for each tree ∈ new_theory do
          tree ← EXPAND_NODES(tree)
      end for
      Compute score theory and new_theory
      if score_new_theory > score_theory then
          return new_theory
      else
          return theory
      end if
  end function
```

Next, we provide more details about the revision operators devised in this work.

Pruning is a technique that reduces the size of trees by removing nodes of the tree where the wrong predictions lie. The pruning operator has two primary goals: (1) to cover more examples along a path, which is the equivalent of generalizing clauses, by removing nodes (literals) possibly responsible for wrong predictions; and (2) to reduce the size of the trees which may contribute to three additional benefits: (1) improve the inference time, (2) make the trees more interpretable, and (3) help in the rest of the revision process, since it is also subject to tree depth limitations.

The structure of our pruning algorithm is quite simple: it makes a bottom-up pass through a given tree, and decides, for each node, whether to leave the node as it is, or whether to delete this node and make its parent become a leaf. The decision is made considering the success or failure weighted variance of a path ending in a node. Thus,

the algorithm recursively attempts to remove nodes whose children are leaves and revision points, from bottom to up and keeps subtrees that contain at least one path not marked as a revision point.

As mentioned earlier, a node is good enough to stop the growth of its subtree when its weighted variance is less than a given δ. Oppositely, we consider a node not good enough to remain in the tree when its weighted variance is higher than δ. By removing such a node, we are giving a chance for the algorithm to later find a possible expansion of nodes that would result in better splits. We implemented the value of δ as 0.0025, which is the default value for stopping the growth of a subtree in the RDN-Boost algorithm. The Pruning operation is presented as Algorithm 4.

Algorithm 4 Pruning Operator: Removes nodes recursively if they fit the definition of Revision Point

```
function PRUNING(node)
    left ← PRUNING(node.left)
    right ← PRUNING(node.right)
    if left and right child are leaves and both have variance greater than δ then
        Remove node from node and put a leaf in its place
    end if
    return node
end function
```

The *Expansion* operator proceeds by adding nodes in an initial theory. As the initial theory is preferably nonempty, as required by Algorithm 3, this process takes advantage of a starting point, instead of learning from scratch. Adding new nodes and performing splits from starting points may lead to paths that would otherwise not be found in the constrained search space, possibly resulting in better covering. Thus, this process is important for two main reasons: (1) by adding nodes in existing paths, it has the same effect of specializing clauses by adding literals to make them fit more to target data; and (2) it takes advantage of the starting point obtained by transference. The expansion is done similarly to the process of learning from scratch; it considers leaves that still need to grow into subtrees as revision points and searches for the node that gives the best split, according to the weighted variance, as the splitting criterion. The leaves and their regression values are computed when the path is good enough, or the tree has reached the maximum depth or number of clauses. Algorithm 5 presents the procedure used here to perform the expansion of nodes. Figure 3 brings an example of the transfer learning process proposed here from model learned from IMDB and transferred to UW-CSE (more details about these domains can be found in the next section).

Algorithm 5 Performing expansion of nodes

```
function EXPAND_NODES(node)
    left ← left child of node
    if left is a leaf and it has variance greater than δ then
        Find a new node that gives the best split
        Add this best node to left
        left ← EXPAND_NODES(left)
    end if
    right ← right child of node
    if right is a leaf and it has variance greater than δ then
        Find the node that gives the best split
        Add this best node to right
        right ← EXPAND_NODES(right)
    end if
    return node
end function
```

4 Experiments

In this section, we present the experiments we conducted in this paper in order to investigate the following research questions:

- **Q1:** Does TreeBoostler learn more accurate models than the baselines?
- **Q2:** Does the theory revision step improve the performance of the transfer process?
- **Q3:** Does TreeBoostler transfer well across domains?
- **Q4:** Is TreeBoostler faster than the baselines?

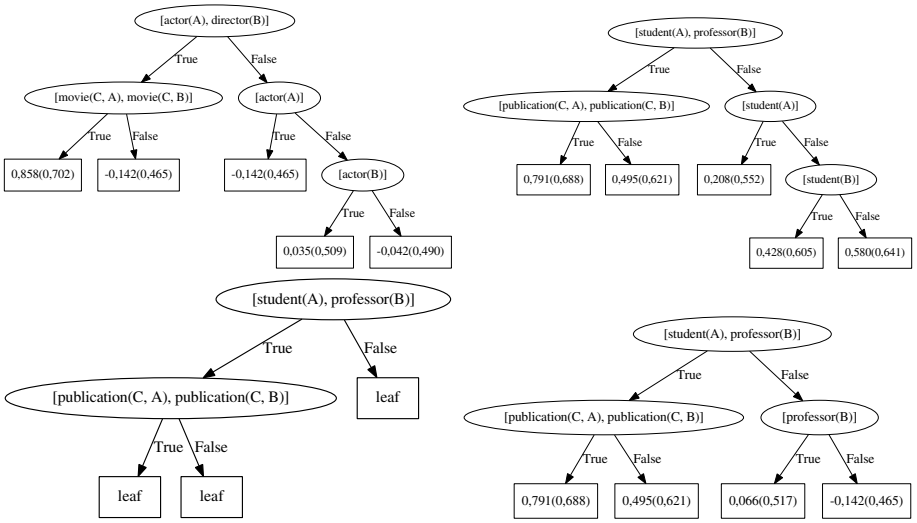

Fig. 3 The transfer learning process stages. The trees presented are the following: obtained from source domain by learning from scratch (top-left); transferred by mapping predicates (top-right); after pruning process (down-left) and after expansion of nodes (down-right). All trees are the first one learned in the iterations. The transference is done from IMDB to UW-CSE and depth limits were reduced to generate smaller trees. Regression values are not considered in pruning process and they are relearned when expanding nodes

- **Q5:** Does TreeBoostler perform better than the baselines with an increasing amount of examples in the target data?
- **Q6:** Does TreeBoostler perform better than the baselines with minimal target data?

We compare the performance of TreeBoostler against three baseline approaches that learn from scratch from target data: RDN-B which learns a set of regression trees using boosting method, RDN which learns a single large regression tree and MLN-B (Khot et al. 2011a, 2015) which uses functional gradient boosting method to learn either a set of trees or clauses to represent a MLN. We also compare TreeBoostler with TODTLER (Van Haaren et al. 2015), a transfer learning method that lifts a source structure to second-order logic. We do not compare against the state-of-the-art algorithm LTL (Kumaraswamy et al. 2015) because the complete transfer and theory refinement system is not available.

Experimental Methodology: To observe if the theory revision stage improves the performance of the whole transfer process, two versions of the algorithm are considered: (1) transference considering predicates mapping and parameter learning only, *i.e.*, the first stage of the complete algorithm (TreeBoostler*) and (2) the complete transfer system using predicate mapping and theory revision (pruning and expansions of trees) (TreeBoostler). For TreeBoostler, we restricted the depth limit of the trees to be 3, the number of leaves to be 8, the number of regression trees was 10, and the maximum number of literals per node to 2. We used the same settings to learn from scratch using the method RDN-B. For the single tree RDN method, we used 20 leaves. For training all the RDN based algorithms, we subsampled the negative examples in a ratio of two negatives for one positive. Thus, following Natarajan et al. (2012), we set the initial potential to be -1.8. For testing, we presented all the negative examples. For MLN-B, we used the clause-based representation with default settings. For the MLN-based transfer approach TODTLER, we used Alchemy with default settings and MC-SAT algorithm (option *-ms*) to compute the probabilities. Also, we kept the default parameters and generated second-order templates containing at most three literals and three object variables.

Datasets: Following the previous literature we present our results considering seven publicly available datasets described as follows.

- The Cora dataset (Bilenko and Mooney 2003) is a database of computer science citations with 1295 different citations to 112 computer science research papers and has fields as author, venue, title and year. We consider two tasks in this dataset: to predict the *sametitle* and the *samevenue* relations. This dataset is divided into five mega-examples.
- The WebKB dataset (Craven and Slattery 2001) consists of labeled web pages from computer science departments of four universities. The dataset contains information about links between web pages, classification of web pages, words that appears on the web pages, instructors, and members of projects. The goal in this dataset is to predict the *departmentof* relation that determines the department of a given web page. This dataset is divided into four mega-examples, one for each university.
- The UW-CSE dataset (Khosravi et al. 2012) consists of information about professors, students, and courses from 5 different areas of computer science (artificial intelligence, programming languages, theory, system, and graphics). Thus, this dataset is divided into five mega-examples according to the mentioned areas. It includes predicates that represent publications and their authors, projects and their members, level of courses, and so forth. The goal is to predict the *advisedby* relation that identifies a student being advised by a professor.

- IMDB (Mihalkova and Mooney 2007) is a dataset that describes a movie domain and presents predicates as director, actor, genre, movie, among others, and the relationships between them. It is divided into five mega-examples where each one contains information about four movies. The goal is to predict the *workedunder* relation that identifies an actor that has worked for a director.
- The Yeast protein (Mewes et al. 1999) dataset is obtained from MIPS[1] Comprehensive Yeast Genome Database and includes information about proteins with their location, function, phenotype, enzyme, among others. The goal in this dataset is to predict the class of a protein. This dataset contains four folds independent of each other.
- Twitter (Van Haaren et al. 2015) is a dataset that contains tweets about Belgian soccer matches. The information is words that are tweeted, relations between accounts (following relation) and the type of accounts (club, fan, or news). The goal is to predict the type of account in two independent folds.
- NELL (Carlson et al. 2010) is a machine learning system that extracts probabilistic knowledge base from online text data. We consider two domains from NELL dataset, which are the Sports domain, extracted from the iteration 1070 and the Finances domain, extracted from the iteration 1115. The goal in the Sports domain is to predict the relation that defines a team playing a sport. The Finances domain has a goal of predicting the relation that defines a company belonging to an economic sector. In order to obtain different folds, we split the data of the target predicate randomly into three parts. Thus, each fold consists of parts of the target predicates and all facts (non-target predicates).

Experiments For all the experiments, we allowed TreeBoostler to search for all permutations of arguments of a given predicate. This action was crucial for transferring among NELL datasets since some source predicates are the inverse of a possible mapped target predicate. Also, we did not allow more than one distinct source predicate to be mapped to the same target predicate, as this bias does not improve the results while still increases the training time. The option *searchEmpty* was also set to false to avoid increasing the amount of training time.

The first experiment simulates the learning process from limited data which is the more suitable scenario for transfer learning. We employed the same methodology used in related works: training is performed on one fold and testing on the remaining $n-1$ folds. The results are then averaged over n runs. For each run, a new learned source model is used for transference. Specifically for TODTLER, the results were obtained from one single run due to extremely time-consuming resources when computing scores for each first-order clause using Alchemy. TODTLER was not able to finish computing scores for clauses in NELL and WebKB datasets after one week and MLB-B was not able to infer the probabilities for the Cora test set in the same amount of time. We used the following measures to compare the performance: conditional log-likelihood (CLL), the area under the ROC curve (AUC ROC), the area under the PR curve (AUC PR) and training time. Note that in the training time of transfer systems, we did not consider the time necessary to learn from the source domain.

The results are presented in the Tables 2, 3, 4, 5 and 6. The Tables 2, 3, and 4 present the transfer experiments for the pairs of datasets IMDB and Cora and also Yeast and

[1] Munich Information Center of Protein Sequence.

Table 2 Results on IMDB and Cora (sametitle) dataset

Algorithm	IMDB → Cora (sametitle)				Cora (sametitle) → IMDB			
	CLL	AUC ROC	AUC PR	Time	CLL	AUC ROC	AUC PR	Time
RDN	**−0.211**	0.764	0.073	3.85 s	−0.168	**0.987**	0.720	0.94 s
RDN-B	−0.336	0.821	0.206	32.36 s	**−0.075**	**1.000**	**0.986**	2.44 s
MLN-B	NA	NA	NA	NA	−0.262	**0.998**	**0.905**	3.96 s
TODTLER	−4.454	0.504	0.458	27 min	−0.923	0.885	0.537	197.77 s
TreeBoostler*	**−0.244**	0.721	0.476	1.84 s	−0.307	0.868	0.092	1.48 s
TreeBoostler	**−0.232**	**0.888**	**0.599**	30.17 s	**−0.075**	**1.000**	**0.979**	6.73 s

We compare our algorithm, RDN-B (that uses boosting), MLN-B (clause-based representation), RDN and TODTLER. We present the results for the area under curves for ROC and PR and the conditional log-likelihood for test examples. We also present the training time

Table 3 Results on IMDB and Cora (samevenue) dataset

Algorithm	IMDB → Cora (samevenue)				Cora (samevenue) → IMDB			
	CLL	AUC ROC	AUC PR	Time	CLL	AUC ROC	AUC PR	Time
RDN	**−0.192**	0.641	0.074	16.70 s	−0.166	**0.994**	0.813	1.17 s
RDN-B	−0.277	**0.842**	**0.270**	237.47 s	−0.073	**1.000**	**1.000**	3.29 s
MLN-B	NA	NA	NA	NA	−0.434	**0.997**	0.879	3.77 s
TODTLER	−5.213	0.519	**0.371**	17 min	−0.923	0.885	0.537	195.77 s
TreeBoostler*	−0.323	0.582	0.183	5.34 s	−0.213	0.958	0.727	2.82 s
TreeBoostler	−0.298	**0.707**	**0.292**	106.39 s	−0.077	**0.999**	**0.952**	10.70 s

We compare TreeBoostler algorithm, RDN-B (that uses boosting), MLN-B (clause-based representation), RDN and TODTLER. We present the results for the area under curves for ROC and PR and the conditional log-likelihood for test examples. We also present the training time

Table 4 Results on Yeast and Twitter dataset

Algorithm	Yeast → Twitter				Twitter → Yeast			
	CLL	AUC ROC	AUC PR	Time	CLL	AUC ROC	AUC PR	Time
RDN	**−0.155**	**0.964**	0.271	4.08 s	**−0.182**	0.695	0.081	4.46 s
RDN-B	**−0.118**	**0.993**	0.382	24.42 s	−0.257	0.919	**0.231**	18.80 s
MLN-B	**−0.249**	0.819	0.312	114.10 s	−0.288	0.674	0.154	9.68 s
TODTLER	−1.259	0.520	0.368	13.42 s	−0.023	0.497	0.002	39 min
TreeBoostler*	**−0.138**	**0.986**	0.394	6.12 s	**−0.180**	**0.986**	**0.273**	4.14 s
TreeBoostler	**−0.118**	**0.993**	0.362	114.71 s	**−0.180**	**0.986**	**0.272**	60.99 s

We present the results for the area under curves for ROC and PR, the conditional log-likelihood and the training time

Twitter. Each dataset was treated as a source domain and target domain. The Table 5 presents the transfer experiments UW-CSE → WebKB and NELL Sports → NELL Finances. We omitted the opposite transfer experiments because transferring from WebKB to

Table 5 Results on transference from UW-CSE to WebKB dataset and NELL sports domain to finances domain considering area under the curves for ROC and PR, the conditional log-likelihood and the training time

Algorithm	UW-CSE → WebKB				NELL Sports → NELL Finances			
	CLL	AUC ROC	AUC PR	Time	CLL	AUC ROC	AUC PR	Time
RDN	− 0.141	0.571	0.032	110.87 s	− 0.180	0.532	0.020	4.59 s
RDN-B	**− 0.081**	0.801	0.138	13 min	− 0.317	0.713	**0.083**	22.12 s
MLN-B	− 0.203	**0.995**	**0.414**	113 min	− 0.205	0.503	0.007	16.22 s
TODTLER	NA	NA	NA	NA	NA	NA	NA	NA
TreeBoostler*	− 0.287	0.888	0.013	11.74 s	**− 0.164**	**0.978**	**0.062**	46.63 s
TreeBoostler	**− 0.080**	**0.916**	**0.292**	19 min	**− 0.161**	**0.979**	**0.074**	229.36 s

Table 6 Results on transference from IMDB to UW-CSE dataset considering area under the curves for ROC and PR, the conditional log-likelihood and the training time

Algorithm	IMDB → UW-CSE			
	CLL	AUC ROC	AUC PR	Time
RDN	**− 0.194**	0.918	**0.247**	1.79 s
RDN-B	**− 0.261**	**0.935**	**0.265**	8.17 s
MLN-B	− 0.707	0.893	0.152	4.19 s
TODTLER	− 3.699	0.570	0.037	208 min
TreeBoostler*	**− 0.274**	0.926	**0.275**	1.16 s
TreeBoostler	**− 0.241**	**0.940**	**0.305**	9.20 s

UW-CSE is too easy and it was not possible to map the predicates from NELL Finances. The Table 6 presents the transfer experiment IMDB → UW-CSE. It can be observed that our algorithms are competitive or better than TODTLER and learning from scratch methods. Our algorithms and learning from scratch methods outperform TODTLER in most of the results presented, mostly due to the efficiency and expressiveness of the language used for representing RDNs. Therefore, it is more interesting to compare our algorithms against learning from scratch methods. The TreeBoostler algorithm performed comparably or better than learning from scratch methods in all but two experiments for AUC ROC. Even for the TreeBoostler*, which is restricted only for mapping, was able to learn more accurate models than learning from scratch in 2 experiments for AUC ROC and 4 for AUC PR. Then only mapping the predicates and learning the parameters for the mapped trees may be very useful when target training data is scarce. The most significant result can be observed in the transference from the real-world dataset NELL Sports to NELL Finances. The values in bold highlight the best results which are significantly better than the performance of the remaining algorithms, but not significantly better concerning one another. Based on these experiments and observations, we can positively answer the questions **Q1** and **Q3** posed before.

As can be seen from the results, the training time consumed by TreeBoostler* is usually smaller than RDN-B and equivalent to RDN. This is because the transfer algorithm only needs to find the best split for those nodes that have not-mapped predicates; otherwise it already knows which mapped node to consider in the split, avoiding searching and evaluating other possible mappings. The first time a predicate appears in the set of regression

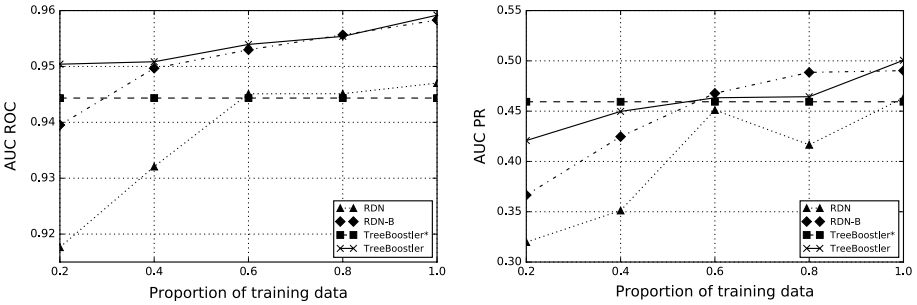

Fig. 4 Learning curves for AUC ROC (left) and AUC PR (right) obtained from IMDB → UW-CSE

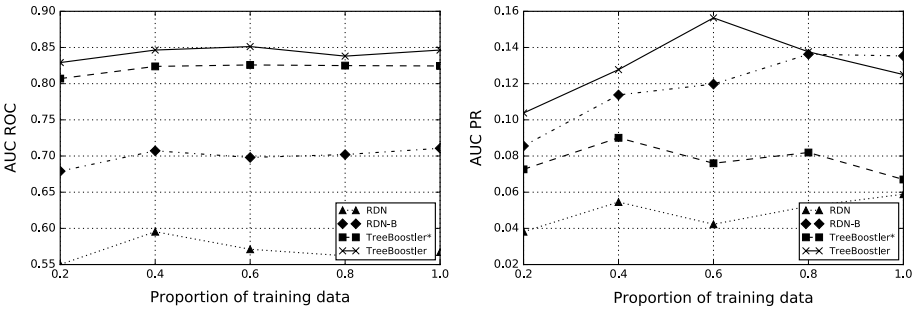

Fig. 5 Learning curves for AUC ROC (left) and AUC PR (right) obtained from NELL Sports → NELL Finances

trees is the only time a mapping has to be found for this predicate. It saves time in the rest of the tree and the next iterations as the algorithm knows how to transfer an inner source node. On the other hand, TreeBoostler, considering Theory Revision, improves accuracy but is computationally costly since it is another search approach. This training time considers the time spent in the entire process which includes the time taken for transference, the time taken for evaluating both the transferred and the revised model, and the time taken for pruning and expansion. In summary, we can answer **Q4** affirmatively for TreeBoostler* and affirmatively comparing to other transfer learning system for TreeBoostler. The results show that question **Q2** can also be answered positively. The Theory Revision process shows an improvement in the performance for all the metrics except for a worse AUC PR in a single experiment.

In order to compare the performance of TreeBoostler method with increasing amounts of target data, we performed a learning curve experiment to transfer some of the same pairs of datasets. For these experiments, we employed the traditional cross-validation methodology when training is performed on $n-1$ folds and testing on the remaining one fold. The data selected for training is then shuffled and divided into five sequence parts. All systems observed the same sequence of these parts. The entire process is done in n runs, and the curves are obtained by averaging the results. Figures 4, 5, 6, 7, 8, 9, 10 and 11 demonstrate this experiment. As can be seen, our algorithm outperforms or equates learning from scratch RDN-B in most of the results, particularly with smaller amounts of data (about 40% of the target data). One exception is the learning curve for the AUC ROC in

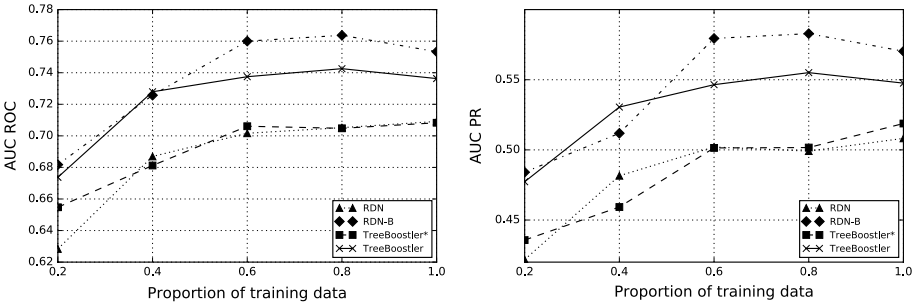

Fig. 6 Learning curves for AUC ROC (left) and AUC PR (right) obtained from Yeast → Twitter

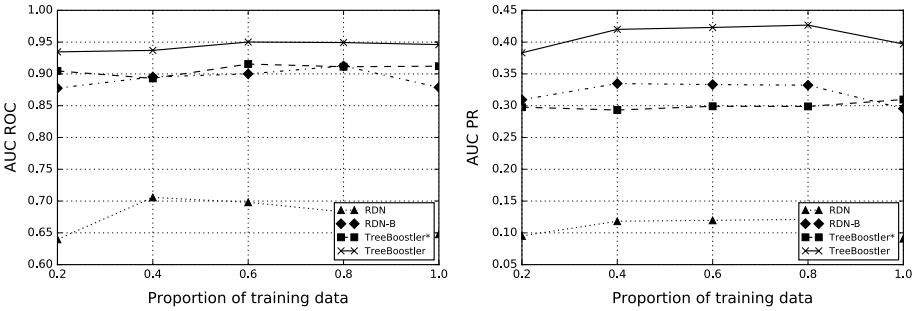

Fig. 7 Learning curves for AUC ROC (left) and AUC PR (right) obtained from Twitter → Yeast

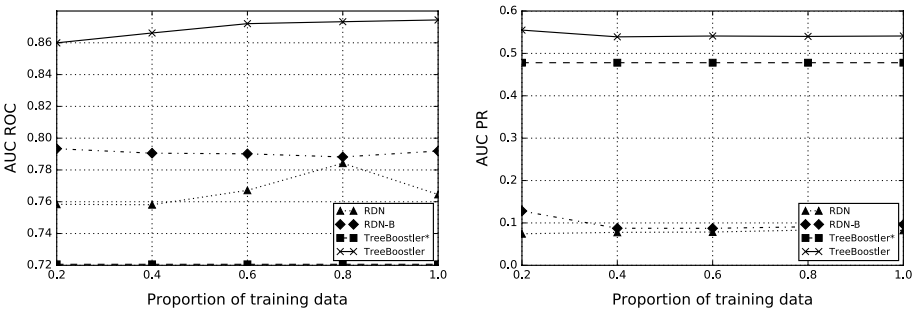

Fig. 8 Learning curves for AUC ROC (left) and AUC PR (right) obtained from IMDB → Cora (sametitle)

Fig. 9, which demonstrates a decreasing in the performance as the target data increases. In this experiment, TreeBoostler is outperformed by RDN-B until 80% of the target data, although it outperforms RND-B in terms of AUC PR. Thus, question **Q5** can be answered affirmatively.

A third experiment was conducted in order to address the problem of minimal target data and investigate how the algorithms behave when learning from only a few examples. We also performed a learning curve experiment with the same pairs of datasets. We employed the traditional cross-validation methodology, then we shuffled the data for training and selected five groups of 5 positive examples and five groups of 5 negative

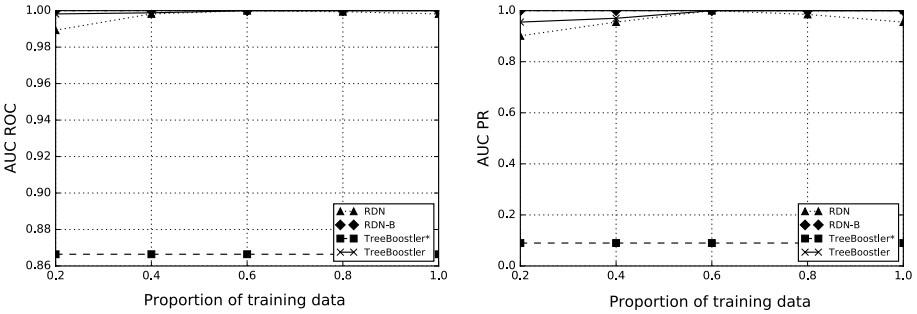

Fig. 9 Learning curves for AUC ROC (left) and AUC PR (right) obtained from Cora (sametitle) → IMDB

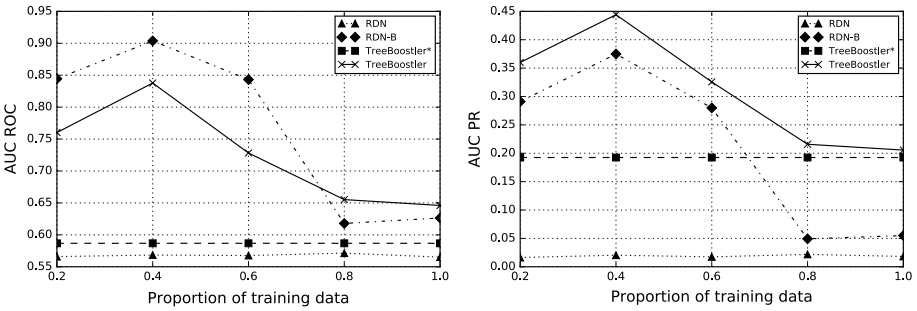

Fig. 10 Learning curves for AUC ROC (left) and AUC PR (right) obtained from IMDB → Cora (samevenue)

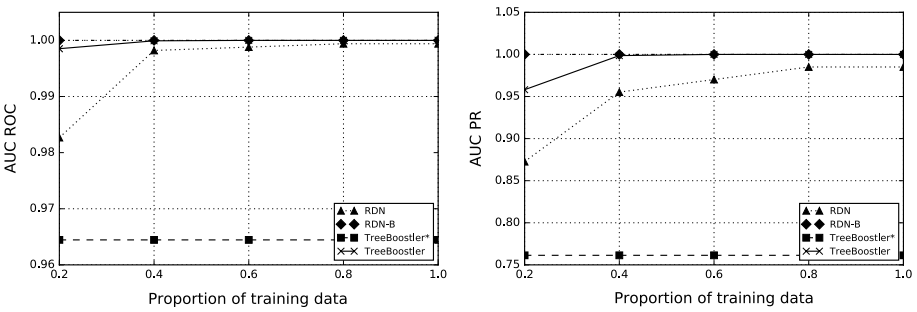

Fig. 11 Learning curves for AUC ROC (left) and AUC PR (right) obtained from Cora (samevenue) → IMDB

examples. All systems observed the same sequence of these groups of examples, *i.e.*, systems observed from 5 up to 25 examples for each label. Similarly to the last experiment, the entire process is done in n runs, and the curves are obtained by averaging the results. Figures 12, 13, 14, 15, 16, 17 and 18 demonstrate this experiment. As indicated in the experiments, TreeBoostler easily outperforms the learning from scratch algorithms RDN-B and RDN in all the presented results. The small amount of training data available was insufficient to learn good models in the learning from scratch

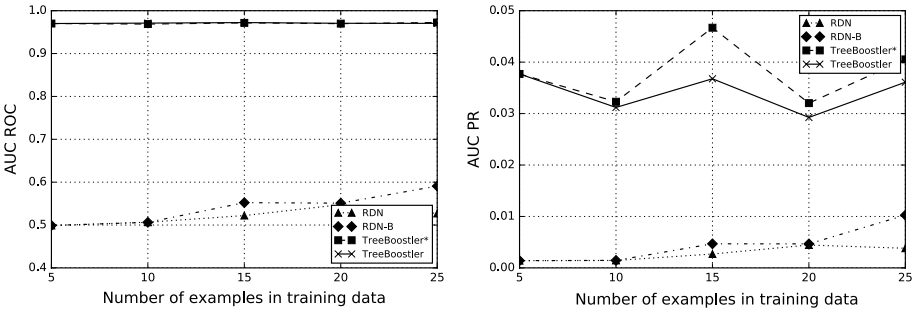

Fig. 12 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from NELL Sports → NELL Finances

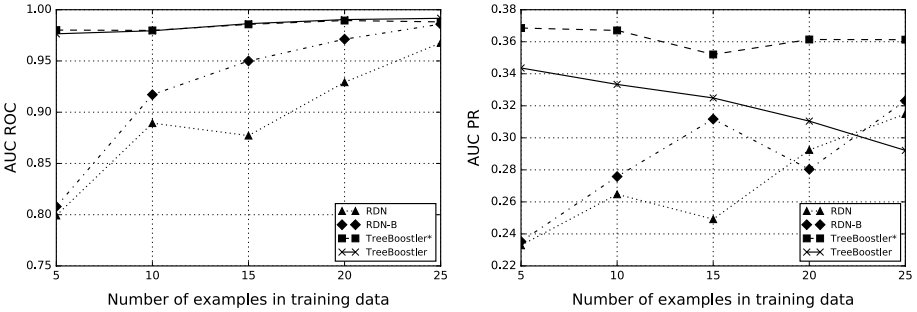

Fig. 13 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Yeast → Twitter

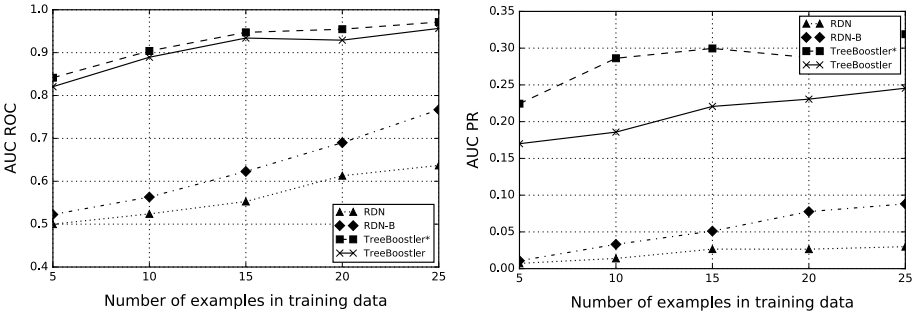

Fig. 14 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Twitter → Yeast

approaches, especially for NELL Finances and Yeast datasets. Providing more examples has shown to increase the performance of these approaches; however, it was still insufficient compared to TreeBoostler, which also increased its performance with more examples. As can be seen, the revision step also showed to slightly decrease the performance in the experiments, except for the experiments in Figs. 15, 16, 17 and 18 . This may be

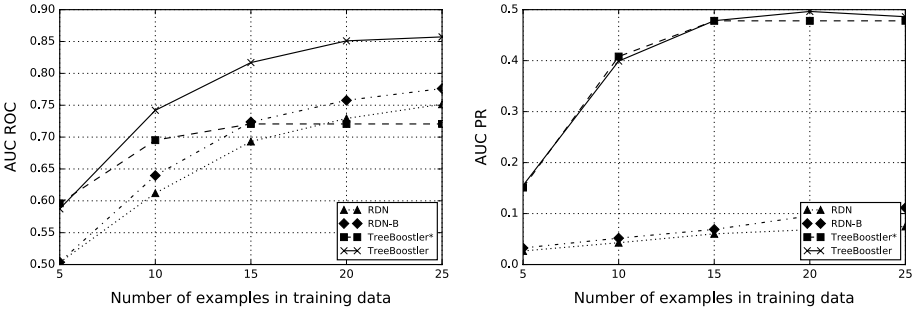

Fig. 15 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from IMDB → Cora (sametitle)

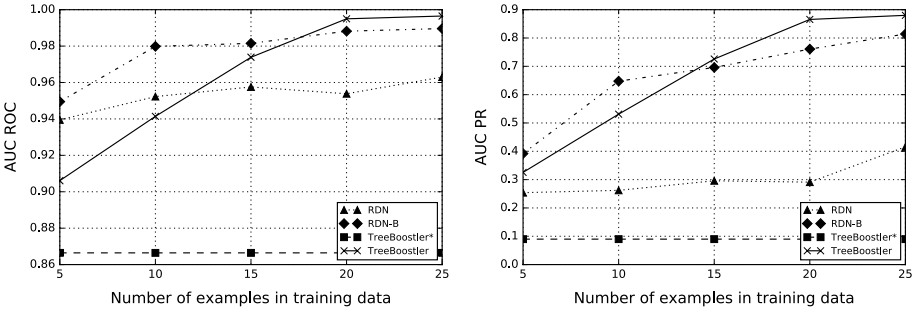

Fig. 16 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Cora (sametitle) → IMDB

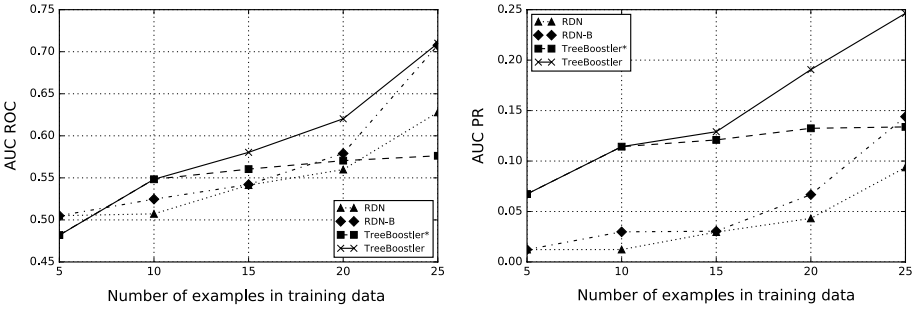

Fig. 17 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from IMDB → Cora (samevenue)

basically due to difficulty of revising and simultaneously relearning parameters of models given very few examples. Since the pruning and expansion operators are subject to the threshold δ, very few examples may not be sufficient to determine correctly when a node is "badly" predicting. Thus, according to these experiments, we can answer question **Q6** positively.

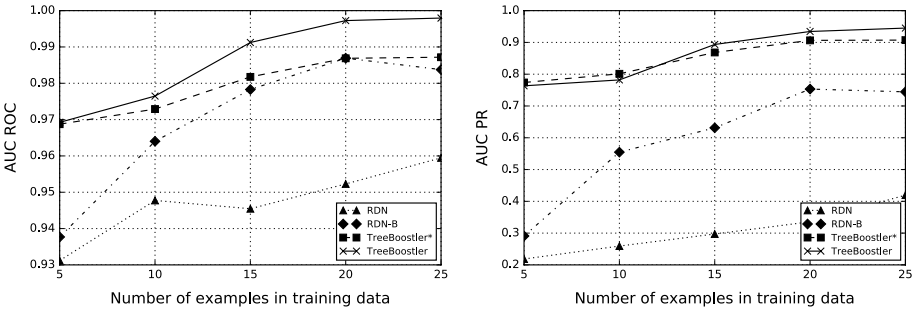

Fig. 18 Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Cora (samevenue) → IMDB

5 Conclusions

In this paper, we have presented a complete transfer learning algorithm, named as TreeBoostler, that transfers Boosted RDNs learned from a source domain to a desirable target domain. TreeBoostler constructs a target set of regression trees biased by a predicate mapping found through the transfer process given the structure of the source regression trees. Then, it applies a second stage process relying on Theory Revision, to propose modifications to the mapped model. These modifications are done through two proposed revision operators for the regression trees, which are the pruning operator and the expansion operator. The pruning operator showed to be essential for deleting nodes in the tree and providing space for the expansion of new nodes in the tree. Through experimental results, we found out that even the first state of the entire transfer process, which only maps predicates and learn the parameters of them, can give better results than learning from scratch in a smaller amount of training time.

Our experimental results demonstrate that this algorithm is effective compared to the other transfer algorithm TODTLER mainly because of the efficiency and expressiveness of the language used for representing RDNs. We also showed from experiments that transfer learning potentially results in more accurate models compared to learning from scratch methods. Moreover, the theory revision process, in general, improved the performance of the transferred models showing the effectiveness of proposing modifications to fit the model to the target data. However, there are some cases that transfer from another domain resulted in less accurate models. It remains a future investigation to understand whether or not to transfer from one domain to another. According to the experiments, our algorithm also demonstrated to be as much efficient as learning from scratch methods.

A possible future direction is to take advantage of stochastic search methods (Paes et al. 2017) in the process of pruning, allowing the method to generate different pruning in the trees at random and expand nodes from these candidates. The stochastic search may help the algorithm to escape from a local optimum since the pruning process follows a deterministic criterion. Another possible research direction is developing the transfer learning process to regression trees that contain predicates with constants or numeric values, increasing the expressiveness of the language used for transference. It is also interesting to investigate how to compute the similarity between domains beforehand to avoid a negative transfer and to enhance the predicate mapping. Finally, we believe that the framework proposed here could be made general enough to be employed to other SRL methods, as long

as they can be boosted. Verifying this belief, both theoretically and empirically is a promising direction for future research.

Acknowledgements We would like to thank the Brazilian Research Agencies CAPES, CNPq (421608/2018-8 (AP) and 305198/2017-3 (GZ)), and FAPERJ (E-26/202.914/2019 (AP)) for the financial support. We also express our gratitude to the authors of RDNBoost and TODTLER for making their code available to us and help us with some doubts, the authors of the datasets that we used in this paper, and the anonymous reviewers for their valuable feedback.

References

Bilenko, M., & Mooney, R. J. (2003). Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, ACM, KDD '03* (pp. 39–48).

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*(1–2), 285–297.

Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka, E. R. Jr., & Mitchell, T. M. (2010). Toward an architecture for never-ending language learning. In *Proceedings of the twenty-fourth AAAI conference on artificial intelligence, AAAI'10* (pp. 1306–1313). Palo Alto: AAAI Press.

Craven, M., & Slattery, S. (2001). Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, *43*(1), 97–119.

Dai, W., Yang, Q., Xue, G. R., & Yu, Y. (2007). Boosting for transfer learning. In *Proceedings of the 24th international conference on machine learning, ACM, ICML'07* (pp. 193–200).

Davis, J., & Domingos, P. (2009). Deep transfer via second-order markov logic. In *Proceedings of the 26th international conference on machine learning (ICML-09)*.

Duboc, A. L., Paes, A., & Zaverucha, G. (2009). Using the bottom clause and mode declarations in FOL theory revision from examples. *Machine Learning*, *76*(1), 73–107.

Friedman, J. H. (2000). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, *29*, 1189–1232.

Getoor, L., & Taskar, B. (2007). *Introduction to statistical relational learning*. *Adaptive computation and machine learning*. MIT Press.

Gutmann, B., & Kersting, K. (2006). Tildecrf: Conditional random fields for logical sequences. In *Proceedings of machine learning: ECML 2006, 17th European conference on machine learning, Berlin, Germany, September 18–22, 2006* (pp. 174–185).

Heckerman, D., Chickering, D. M., Meek, C., Rounthwaite, R., & Kadie, C. (2001). Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, *1*, 49–75.

Khosravi, H., Schulte, O., Hu, J., & Gao, T. (2012). Learning compact Markov logic networks with decision trees. *Machine Learning*, *89*(3), 257–277.

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J. (2011a). Learning Markov logic networks via functional gradient boosting. In *Proceedings of the 2011 IEEE 11th international conference on data mining, IEEE computer society, ICDM '11* (pp. 320–329).

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J. W. (2011b). Learning Markov logic networks via functional gradient boosting. In *11th IEEE international conference on data mining, ICDM 2011, IEEE computer society* (pp. 320–329).

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J. (2015). Gradient-based boosting for statistical relational learning: The Markov logic network and missing data cases. *Machine Learning*, *100*(1), 75–100.

Kumaraswamy, R., Odom, P., Kersting, K., Leake, D., & Natarajan, S. (2015). Transfer learning via relational type matching. In *Proceedings of the 2015 IEEE international conference on data mining (ICDM), IEEE computer society, ICDM '15* (pp. 811–816).

Lee, JW., & Giraud-Carrier, C. (2007). Transfer learning in decision trees. In *2007 International joint conference on neural networks* (pp. 726–731).

Lee, K., Caverlee, J., & Webb, S. (2010). Uncovering social spammers: Social honeypots + machine learning. In *Proceedings of the 33rd international ACM SIGIR conference on research and development in information retrieval, ACM* (pp. 435–442).

Mewes, H. W., Heumann, K., Kaps, A., Mayer, K., Pfeiffer, F., Stocker, S., et al. (1999). MIPS: A database for genomes and protein sequences. *Nucleic Acids Research*, *27*(1), 44–48.

Mihalkova, L., Huynh, T., & Mooney, R. J. (2007). Mapping and revising markov logic networks for transfer learning. In *Proceedings of the 22nd national conference on artificial intelligence—vol. 1, AAAI'07* (pp. 608–614). Palo Alto: AAAI Press.

Mihalkova, L., & Mooney, R. (2009). Transfer learning from minimal target data by mapping across relational domains. In *Proceedings of the 21st international joint conference on artificial intelligence (IJCAI-09), Pasadena, CA* (pp. 1163–1168).

Mihalkova, L., & Mooney, R. J. (2007). Bottom-up learning of Markov logic network structure. In *Proceedings of 24th international conference on machine learning (ICML-2007)*.

Natarajan, S., Khot, T., Kersting, K., Gutmann, B., & Shavlik, J. (2012). Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, *86*(1), 25–56.

Neville, J., & Jensen, D. D. (2007). Relational dependency networks. *Journal of Machine Learning Research*, *8*, 653–692.

Neville, J., Jensen, D., Friedland, L., & Hay, M. (2003). Learning relational probability trees. In *Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, ACM, KDD '03* (pp. 625–630).

Odom, P., Kumaraswamy, R., Kersting, K., & Natarajan, S. (2016). Learning through advice-seeking via transfer. In *Inductive logic programming—26th international conference, ILP 2016, London, UK, September 4–6, 2016, revised selected papers* (pp. 40–51).

Paes, A., Zaverucha, G., & Costa, V. S. (2017). On the use of stochastic local search techniques to revise first-order logic theories from examples. *Machine Learning*, *106*(2), 197–241.

Pan, S. J., & Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(10), 1345–1359.

Ramon, J., Driessens, K., & Croonenborghs, T. (2007). Transfer learning in reinforcement learning problems through partial policy recycling. In *Machine learning: ECML 2007, 18th European conference on machine learning, proceedings. Lecture notes in computer science* (vol. 4701, pp. 699–707). Berlin: Springer.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*(1–2), 107–136.

Sinapov, J., & Stoytchev, A. (2011). Object category recognition by a humanoid robot using behavior-grounded relational learning. In *2011 IEEE international conference on robotics and automation (ICRA), IEEE* (pp. 184–190).

Van Haaren, J., Kolobov, A., & Davis, J. (2015). TODTLER: Two-order-deep transfer learning. In *Proceedings of the twenty-ninth AAAI conference on artificial intelligence*.

Wrobel, S. (1996). First order theory refinement. In *Advances in inductive logic programming*. Amsterdam: IOS Press.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.