Transfer Learning by Mapping and Revising Boosted Relational Dependency Networks

Rodrigo Azevedo Santos
PESC/COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil

Aline Paes
Departamento de Ciência da Computação
Universidade Federal Fluminense
Niteroi, Brazil

Gerson Zaverucha
PESC/COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil

***Abstract*—Statistical machine learning algorithms usually assume that there is considerably-size data 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 environment. 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 among 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.**

***Index Terms*—transfer learning, statistical relational learning**

- Student level: MSc
- Date of conclusion: June, 2019
- Examining board members: Prof. Gerson Zaverucha, Ph.D. (UFRJ); Prof. Aline Marins Paes Carvalho, D.Sc. (UFF); Prof. Valmir Carneiro Barbosa, Ph.D. (UFRJ); Prof. Fabio Gagliardi Cozman, Ph.D. (USP)
- An on-line version of the full dissertation [1] is available in PESC/COPPE and a publication [2] derived from the research work is available in Machine Learning journal. Accepted at CTD-SBC 2020 [3].

I. Introduction

Machine learning algorithms have been widely and successfully used in many areas such as computer vision, robotics, social network analysis, and others [1], [2]. However, this 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. This is often the case of several real-world problems where collecting data is expensive or even impossible to obtain, such as collecting data from movements of real-world robots [3], collecting WiFi signal data from a large number of locations [4] and labeling data for sentiment classification [5]. To handle this issue, transfer learning techniques [6] leverage a model learned from a source domain with more examples to learn from another, related, domain where data is more scarce.

Transfer learning has been widely employed in classical machine learning settings, such as ensembles [7] and decision trees [8]. 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 of a number of real-world data such as interaction between proteins [9] and interaction between accounts in social media [10]. In addition, real-world data have noise and are generally uncertain. This is the focus of the area called Statistical Relational Learning (SRL) [11]. Transfer Learning algorithms have also been developed in the context of SRL. Two of these algorithms [10], [12] transfer relational knowledge by creating a second-order representation of formulas from learned Markov Logic Networks (MLN) [13]. Other three algorithms [14]–[16] find predicate mappings through search methods to perform transference of clauses learned from MLNs by mapping their predicates.

Although these methods showed better results compared to MLN models learned from scratch, [17] have shown that applying a boosted approach to learn Relational Dependency Networks (RDN) yielded superior performance over traditional SRL approaches. Based on the predicate mapping algorithm presented by [14] to transfer MLN clauses, we developed a similar predicate mapping approach to perform transference of Boosted RDNs, which are models that have a higher expressiveness. We have opted for using Boosted RDNs as the models to be transferred due to its efficiency in both training and inference time and the capability of learning both the structure and the parameters of RDNs simultaneously, which is not the case of the MLN models used by related work. RDN-Boost has been shown to have state-of-the-art results in learning RDNs and superior performance over other SRL models in much less training time.

In this dissertation, motivated by the need of learning from

[1] https://www.cos.ufrj.br/index.php/pt-BR/publicacoes-pesquisa/details/15/2903

[2] https://doi.org/10.1007/s10994-020-05871-x

[3] https://sol.sbc.org.br/index.php/ctd/article/view/11371

scarce, relational and uncertain data, we present a transfer learning algorithm called TreeBoostler that transfers Boosted RDNs by mapping the predicates appearing in the trees. At a higher level, the algorithm generates the possible predicate mappings as it tries to recursively transfer nodes from the source regression trees. After finding such mappings, they are propagated to the rest of the tree and the other trees of the next iterations. To complement the process and better adjust the mapped trees to the new, target domain, TreeBoostler also includes a theory revision [18] algorithm for proposing modifications to the mapped models in order to handle incorrectness and to improve the performance.

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. Our results demonstrate that our method has successfully transferred learned knowledge across different domains in a smaller time compared to other transfer learning algorithms. In addition, transference showed to be very useful in terms of accuracy compared to learning from scratch methods based on RDNs. Additional experiments were performed to investigate the behavior of the algorithm as the number of examples increases and when provided minimal target data. The results demonstrate that our algorithm can be very competitive to traditional methods that learn from scratch even with the increase of the amount of data, also when provided only a few examples.

II. Contributions

To sum up, the main contributions of this dissertation include:

- A transfer learning algorithm, namely TreeBoostler, that constructs a target set of relational regression trees biased by a predicate mapping found through the transfer process given the structure of the source regression trees. This is found by applying all legal mappings to a node and selecting the one which gives the best split.
- A revision theory system that proposes modifications to boosted trees through two revision operators. These revision operators are: (1) pruning operator, which deletes nodes from a tree and (2) expansion operator, which expandes nodes in each tree.
- Three types of experiments to evaluate TreeBoostler against baseline approaches. The experiments were conducted as follows: (1) simulating a transfer learning environment with limited target data, (2) providing to the system a scenario with increasing amounts of target data and (3) providing a scenario with learning from minimal target data.

III. TreeBoostler: The proposed algorithm

We propose a method that transfers learned boosted trees from a source domain to a target domain. The approach is divided into two major steps: first, the source boosted trees structure is transferred to the target domain by finding an adequate predicate mapping, and second, the algorithm revises its trees by pruning and expanding nodes in order to better fit the target data. The regression values are learned simultaneously in both steps. Next, we detail each of these steps.

A. 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 as 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 the parameters relearned to fit the target data. [14] introduced two approaches for establishing a predicate mapping regarding MLNs: (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 later case constructs a predicate mapping only for the predicates that appear in a specific clause, separately, independently on how the predicates appearing in the other clauses were 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 considered 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. In addition, 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 found predicates mapping through the entire process of learning trees.

1) Legal mappings: A mapping is legal if each given source predicate is mapped to a compatible target predicate or to 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. Mapping is done 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: $person \rightarrow project$

and *genre* $\rightarrow$ *person*. Since all sequential 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

Require: $srcPreds$, $tarPreds$, $predsMapping$, $typeConstraints$
Ensure: $mappingsList$
1: $mappingsList \leftarrow []$
2: Pick the first unmapped source predicate $srcPred$
3: **for** $tarPred \in tarPreds$ **do**
4: **if** $isCompatible(srcPred, tarPred)$ **then**
5: Add this mapping to a copy of $predsMapping$
6: Update a copy of $typeConstraints$
7: Call legal_mappings with new parameters
8: Insert mappings to $mappingsList$
9: **end if**
10: **end for**
11: **return** $mappingsList$

A legal mapping is defined as follows: 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 n. 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 $type_{s_k}$ and Z_i is associated to the type $type_{t_j}$, then either $type_{t_j}$ has not appeared before as the second term of an element in S or $type_{s_k} \rightarrow type_{t_j} \in S$. The set of compatible types starts empty and is iteratively filled in with a type correspondence yielded from a predicate mapping.

Note that the boosted trees are learned with respect to a query atom; because of that, the transfer algorithm must receive as input the source and target query atoms to start the 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* $\rightarrow$ *person* and the predicate mapping set with *workedunder(A,B)* $\rightarrow$ *advisedby(A,B)*. Table I shows the final predicate mapping set, found after transferring the entire boosted tree structure.

TABLE I
FOUND PREDICATE MAPPING FOR TRANSFERRING IMDB→UW-CSE.

workedunder(A,B)	$\rightarrow$ advisedby(A,B)
director(A)	$\rightarrow$ professor(A)
actor(A)	$\rightarrow$ student(A)
movie(A,B)	$\rightarrow$ publication(A,B)

2) Finding best mapping and transferring the structure: 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 been translated 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 in order to update the current predicate mapping.

In case the algorithm does not find a compatible mapping, a predicate in the source domain will be 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 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 $\rightarrow$ Cora would result in a null theory as shown in Figure 1 due to the fact that Cora dataset has no unary predicates and the root nodes of learned source trees are conjunctions of unary predicates. To deal with such 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 important 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. 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 save tree depth, the algorithm discards such nodes and promotes the child that covers all examples. The Algorithms 2 and 3 present 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

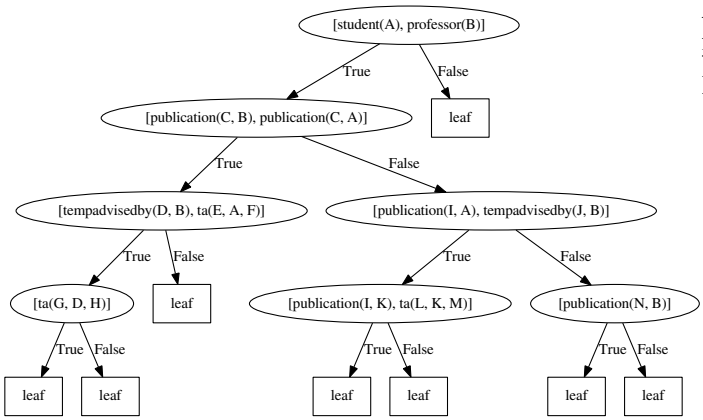

Fig. 1. One regression tree to be transferred from UW-CSE to Cora for query predicate $advisedby$. Regression values are not considered for transference. They are relearned in the process.

allows for example, the mapping of a source predicate with the inverse relation of a target predicate (e.g. *wokedunder(A,B)* $\rightarrow$ *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, the Transfer function

Require: $theory$, a set of regression trees
Ensure: $transferred$, the transferred regression tree
1: $transferred \leftarrow \emptyset$
2: **for** $tree \in theory$ **do**
3: **if** $isCompatible(srcPred, tarPred)$ **then**
4: $newTree \leftarrow \emptyset$
5: TransferTree($tree$, $newTree$)
6: Append $newTree$ to $transferred$
7: **end if**
8: **end for**
9: **return** $transferred$

B. 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 [14]. Such theories may contain multiple faults that prevent them to correctly predict examples due to the difference in the distribution of both domains. These faults can be repaired through the process of theory revision [18]–[20]. The main idea of theory revision is to search for points in the theory that are preventing the examples to be correctly classified and propose modifications to them. In a transfer

Algorithm 3 Top-Level Transfer Algorithm, the TransferTree function

Require: $node$, $transferNode$
1: **if** $node$ is leaf **then**
2: Define $transferNode$ as leaf
3: Stop procedure
4: **end if**
5: $predicates \leftarrow$ Get set of predicates not mapped from $node$
6: **if** $predicates$ is empty **then**
7: $newNode \leftarrow$ Translates predicates in $node$
8: $transferNode \leftarrow newNode$
9: **else**
10: Call LegalMappings given $predicates$ and current predicate mappings and type constraints
11: Generate possible nodes by translating predicates in $node$ according to legal mappings
12: Find the node that gives the best split
13: Update the global variable $predsMapping$ and $typeConstraints$
14: $transferNode \leftarrow$ best node
15: **end if**
16: **if** $transferNode$ is not empty **then**
17: Call TransferTree($node.left$, $transferNode.left$)
18: Call TransferTree($node.right$, $transferNode.right$)
19: **else**
20: **if** $node.left$ is leaf **then**
21: Call TransferTree($node.right$, $transferNode$)
22: **else**
23: Append $node.right$ to to the right-most path of $node.left$
24: Call TransferTree($node.left$, $transferNode$)
25: **end if**
26: **end if**

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 due to the fact that 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 ILP, the points to be changed are defined according to a misclassified example defined according to the proved examples. However, this concept does not hold for the 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.

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 RRTs, 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, as a specialization or a generalization one, 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 considered 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.

Our 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 particularly because the threshold δ can be very sensitive. If this happens, the revision algorithm would 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 in case it has a scoring better than before. The scoring function is the conditional log-likelihood (CLL) over the examples. The Algorithm 4 presents the theory revision process after mapping the vocabulary of the source and target domain.

Algorithm 4 Top-Level Theory Revision Algorithm

Require: $theory$, a set of regression trees
Ensure: $newTheory$, the possibly revised trees
1: $newTheory \leftarrow \emptyset$
2: **for** $tree \in theory$ **do**
3: $newTree \leftarrow$ pruning($tree$)
4: Append $newTree$ to $newTheory$
5: **end for**
6: **if** $newTheory$ is null **then**
7: $newTheory \leftarrow theory$
8: **end if**
9: **for** $tree \in newTheory$ **do**
10: $tree \leftarrow$ ExpandNodes($tree$)
11: **end for**
12: Compute score $theory$ and $newTheory$
13: **if** $score_{newTheory} > score_{theory}$ **then**
14: return $newTheory$
15: **else**
16: return $theory$
17: **end if**

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

1) Pruning: Pruning is a technique that reduces the size of trees by removing nodes of the tree where the bad predictions lie. The pruning operator has two major goals: (1) to cover more examples along a path, which is the equivalent of generalizing clauses, by removing nodes (literals) possibly responsible for bad 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 on 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 successful 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. In the opposite way, we consider a node not good enough to remain in the tree when its weighted variance is greater 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. The Pruning operation is presented as Algorithm 5.

Algorithm 5 Pruning Operator: Removes nodes recursively if they fit the definition of Revision Point

```
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
```

2) Expansion: The Expansion operator proceeds by adding nodes in an initial theory. As the initial theory is preferably nonempty, as required by the Algorithm 4, 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 more fit 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 6 presents the procedure used here to perform the expansion of nodes.

Algorithm 6 Expansion Operator: Performs expansion of nodes

```
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 ← ExpandNodes(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 ← ExpandNodes(right)
end if
return node
```

IV. Experimental Results

We have performed three types of experiments: (1) an approach simulating a transfer learning environment with limited target data, (2) a scenario with increasing amounts of target data and (3) a scenario that represents learning from minimal target data.

A. Research questions

We conducted the experiments in order to investigate the following research questions considering both transfer learning and learning from scratch baselines:

- **Q1:** Does TreeBoostler learn more accurate models than the baselines?
- **Q2:** Does theory revision improve the performance of the transfer process?
- **Q3:** Does TreeBoostler transfer well across domains?
- **Q4:** Is TreeBoostler faster than the baselines?
- **Q5:** Does TreeBoostler perform better than the baselines with increasing amount of examples in the target data?
- **Q6:** Does TreeBoostler perform better than the baselines with minimal target data?

The question **Q1** addresses a common question when comparing different algorithms. It is important to evaluate the algorithm and conclude if it performs better than learning from scratch approaches and related transfer learning approaches. Question **Q2** is important to evaluate the effectiveness of a theory revision process and demonstrates that the process is capable of improving the performance of the transferred model in the target domain. This research question was also addressed in [16]. The question **Q3** addresses if the transfer process is capable of providing good models while question **Q4** asks if the algorithm is faster than related transfer learning algorithms and learning from scratch algorithms. It would be desirable to provide a transfer learning approach that could be faster than learning from scratch. Since the transfer learning system is provided with a trained model from the source domain, part of the time-consuming in the learning process is already done. The question **Q5** addresses how the algorithms behave with increasing amounts of data. This question was also addressed in [10], [12], [14], [16] through learning curves describing the accuracy in different numbers of mega-examples. The question **Q6** addresses the problem of minimal target data where the learner is provided with only a few examples. This problem was the motivation of SR2LR [15] which addressed the *single-entity-centered* setting in which the learner is provided with information concerning only a single entity, i.e. background knowledge contains only information related to the single entity. Differently, in this work, we provided to the learner all the background knowledge available but only a few positive and negative examples.

B. Datasets

Following previous literature, we present our results considering different publicly available datasets described as follows.

- The UW-CSE dataset [21] consists of information about professors, students, and courses from 5 different areas of computer science. The goal is to predict the $advisedby$ relation which identifies a student being advised by a professor.

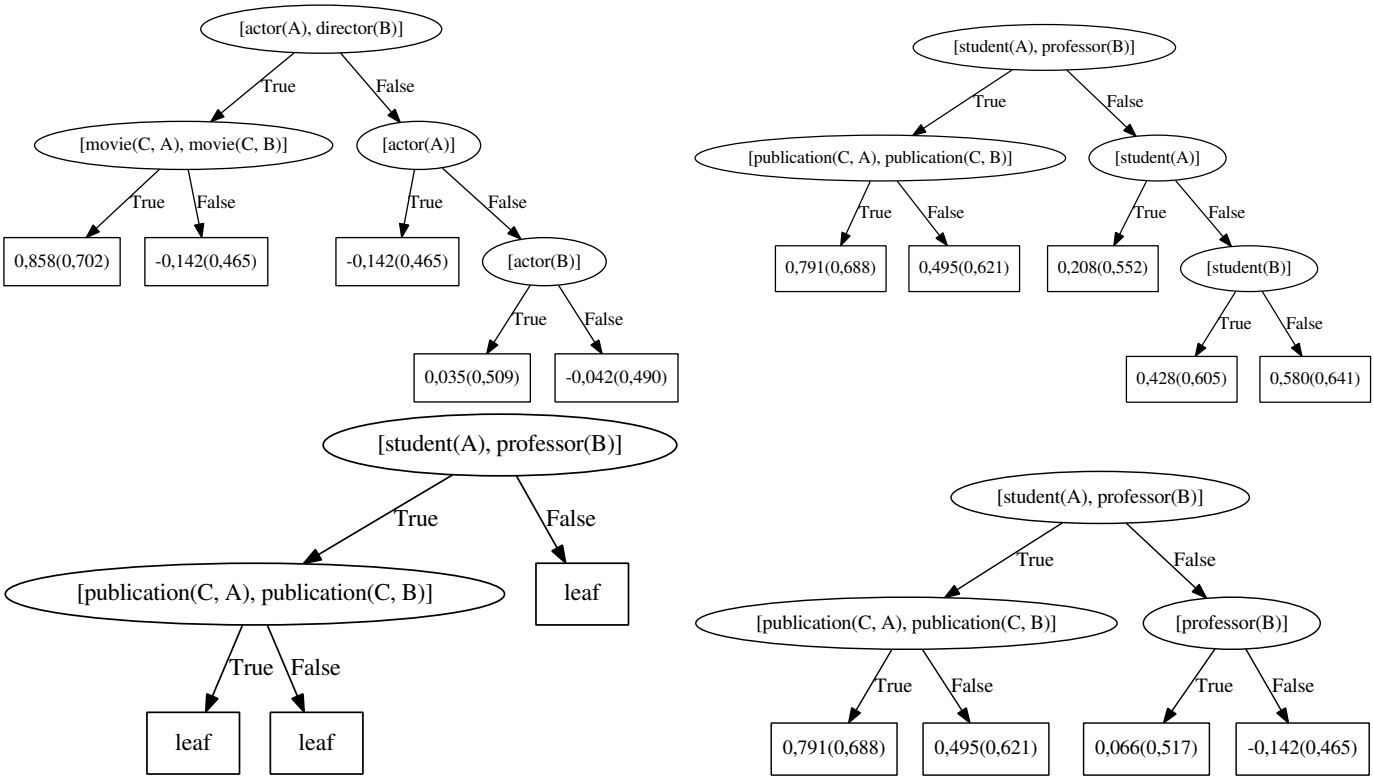

Fig. 2. The transfer learning process stages with their respective trees. Obtained from source domain by learning from scratch (top-left); transferred by mapping predicates (top-right); after the pruning process (down-left) and after the expansion of nodes (down-right).

- IMDB [22] is a dataset that describes a movie domain. The goal is to predict the $workedunder$ relation which identifies an actor that has worked for a director.
- The Yeast protein [9] dataset is obtained from MIPS [4] Comprehensive Yeast Genome Database and includes information about proteins. The goal in this dataset is to predict the class of a protein.
- Twitter [10] is a dataset that contains tweets about Belgian soccer matches. The information is basically words that are tweeted, relations between accounts and the type of accounts. The goal is to predict the type of an account.
- NELL [23] is a machine learning system that extracts probabilistic knowledge base from online text data. We consider only the Sports domain and the Finances domain. The goal is to predict the relation that defines a team playing a sport and to predict the relation that defines a company belonging to an economic sector.

C. Methodology and results

We compared the performance of TreeBoostler against two baseline approaches that learn from scratch from target data: RDN-B, which learns a set of regression trees using boosting method and RDN which learns a single large regression tree. We also compared it against TODTLER [10], a transfer learning method that lifts a source structure to second-order logic.

[4]Munich Information Center of Protein Sequence

To observe if the revision stage improves the performance of the whole transfer process, two stages of the algorithm are considered: transference considering predicates mapping and parameter learning only, i.e. the first stage of the complete algorithm (TreeBoostler*) and 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 was restricted 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 the threshold used in the theory revision step, we set its value as 2.5×10^{-3} which is the same value used as default in the RDN and RDN-B algorithms to decide if a variance of a node is small enough to become a leaf in the learning process. For training all the RDN based algorithms, we subsampled the negatives examples in a ratio of two negatives for one positive. Thus, following [17], we set the initial potential to be -1.8. For testing, we presented all the negative examples. For the MLN based 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.

For all our experiments, we allowed TreeBoostler to search for all permutations of arguments of a given predicate. This

was very important for transferring among NELL datasets since some source predicates can be considered as 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 was done in order to simulate the learning process from very limited data which is the more suitable scenario to transfer learning. We employed the same methodology used in related works [10], [12], [16]: training is performed on 1 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 dataset after one week. 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 II, III, IV and V. 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 one experiment 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 3 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. Bold results are significantly better than the performance of all baselines (RDN, RDN-B and TODLTER) for at least one TreeBostler algorithm. The statistical significance was measured using a paired t-test at the 95% confidence level. Based on these experiments and observations, we can positively answer the questions **Q1** and **Q3** posed before.

As can be seen, 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 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 a source inner 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.

TABLE II
RESULTS ON TRANSFERENCE FROM IMDB TO UW-CSE DATASET.

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
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

TABLE III
RESULTS ON TRANSFERENCE FROM NELL SPORTS DOMAIN TO FINANCES DOMAIN.

Algorithm	NELL Sports → NELL Finances			
	CLL	***AUC ROC***	***AUC PR***	***Time***
RDN	-0.180	0.532	0.020	4.59 s
RDN-B	-0.317	0.713	0.083	22.12 s
TODTLER	NA	NA	NA	NA
TreeBoostler*	-0.164	0.978	0.062	46.63 s
TreeBoostler	**-0.161**	**0.979**	0.074	229.36 s

TABLE IV
RESULTS ON TRANSFERENCE FROM YEAST TO TWITTER DATASET.

Algorithm	Yeast → Twitter			
	CLL	***AUC ROC***	***AUC PR***	***Time***
RDN	-0.155	0.964	0.271	4.08 s
RDN-B	-0.118	0.993	0.382	24.42 s
TODTLER	-1.259	0.520	0.368	13.42 s
TreeBoostler*	-0.138	0.986	0.394	6.12 s
TreeBoostler	-0.118	0.993	0.362	114.71 s

TABLE V
RESULTS ON TRANSFERENCE FROM TWITTER TO YEAST DATASET.

Algorithm	Twitter → Yeast			
	CLL	***AUC ROC***	***AUC PR***	***Time***
RDN	-0.182	0.695	0.081	4.46 s
RDN-B	-0.257	0.919	0.231	18.80 s
TODTLER	-0.023	0.497	0.002	39 min
TreeBoostler*	-0.180	**0.986**	0.273	4.14 s
TreeBoostler	-0.180	0.986	0.272	60.99 s

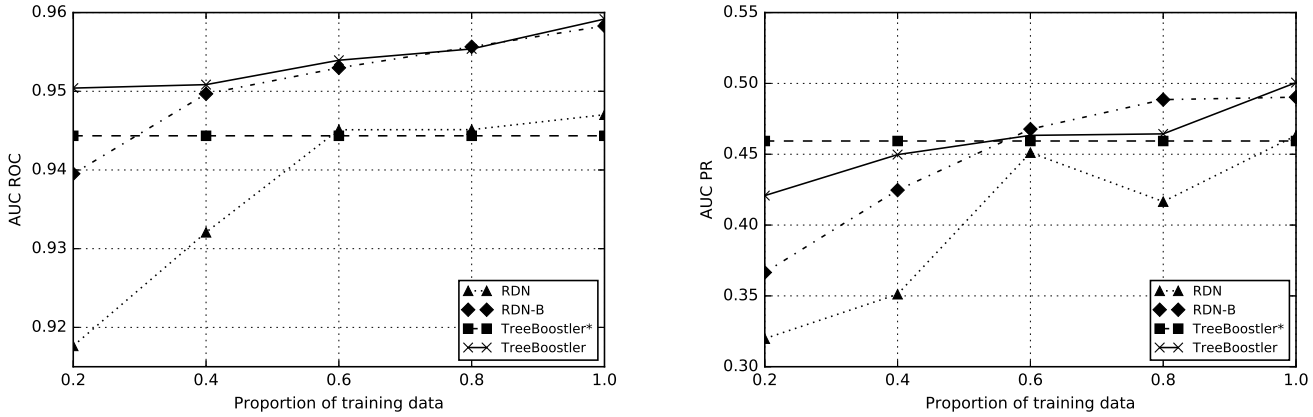

Fig. 3. Learning curves for AUC ROC (left) and AUC PR (right) obtained from IMDB → UW-CSE.

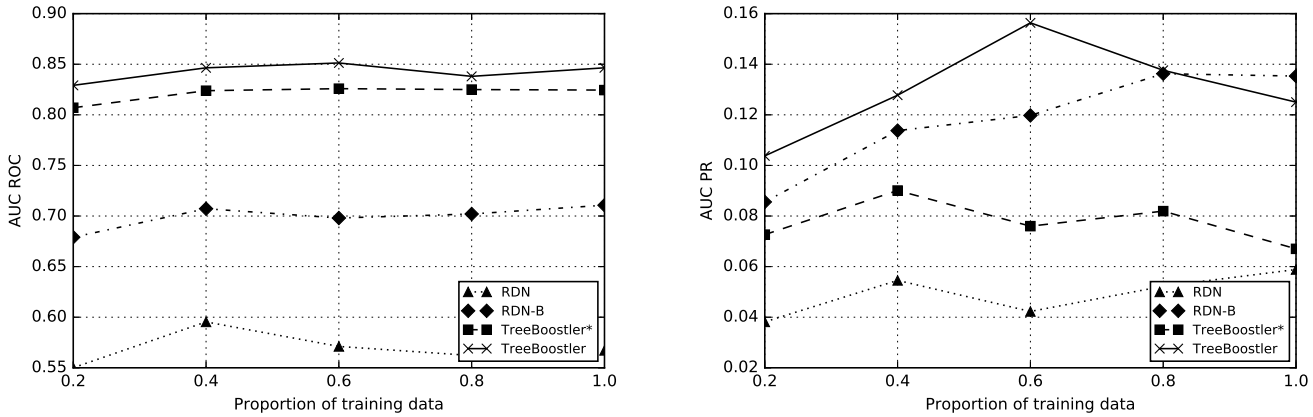

Fig. 4. Learning curves for AUC ROC (left) and AUC PR (right) obtained from NELL Sports → NELL Finances.

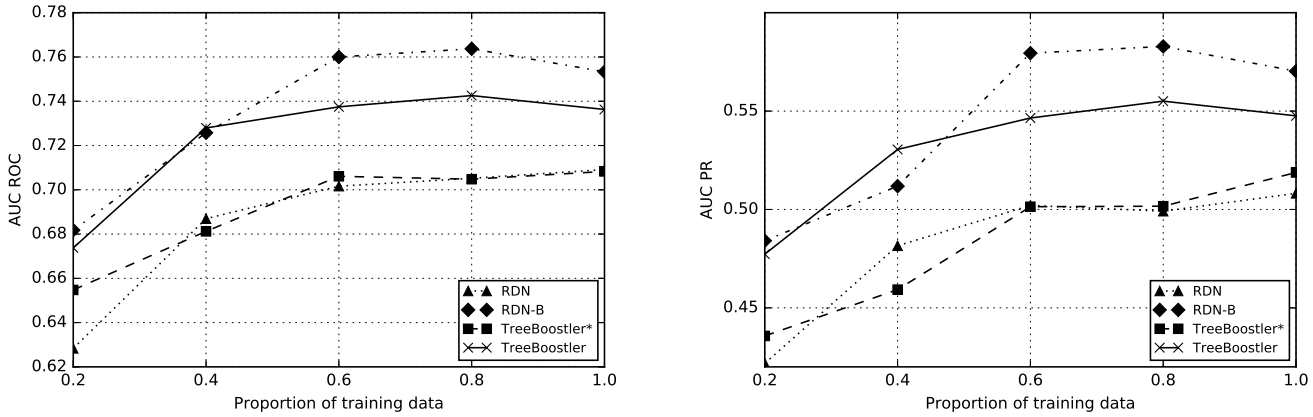

Fig. 5. Learning curves for AUC ROC (left) and AUC PR (right) obtained from Yeast → Twitter.

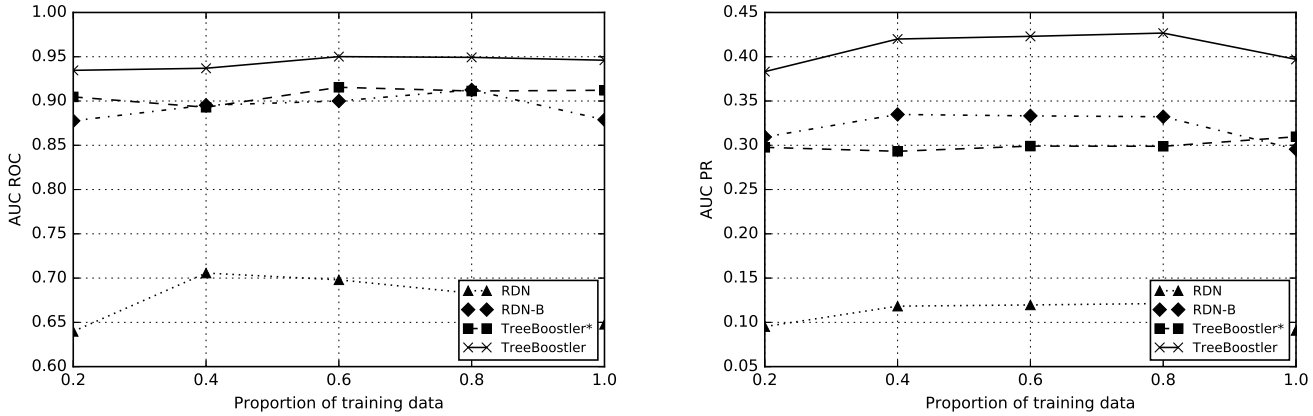

Fig. 6. Learning curves for AUC ROC (left) and AUC PR (right) obtained from Twitter → Yeast.

In order to compare the performance of our method with increasing amounts of target data, we performed a learning curve experiment transferring 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 1 fold. The data selected for training is then shuffled and divided into 5 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. The Figures 3, 4, 5 and 6 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). 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 5 groups of 5 positive examples and 5 groups of 5 negative 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. The Figures 7, 8, 9 and 10 demonstrate this experiment. As indicated in the experiments, our algorithms easily outperform 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 learning from scratch approaches. Providing more examples has shown to increase the performance of these approaches, however it was still insufficient comparing to TreeBoostler which also increased its performance when more examples are provided. As can be seen, the revision step also showed to slightly decrease the performance in the experiments. This may be 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.

V. Conclusion

In this work, we have proposed a novel transfer learning method, 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 important for deleting nodes in the tree and providing space for the expansion of new nodes.

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. The application of theory revision in transfer learning approaches brings the benefit of handling incorrectness that has come from a different yet related domain through transference. Thus, a transference may not provide a good model due to differences between domains. However, this incorrectness may be successfully handled by the modifications proposed in the system.

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 have performed

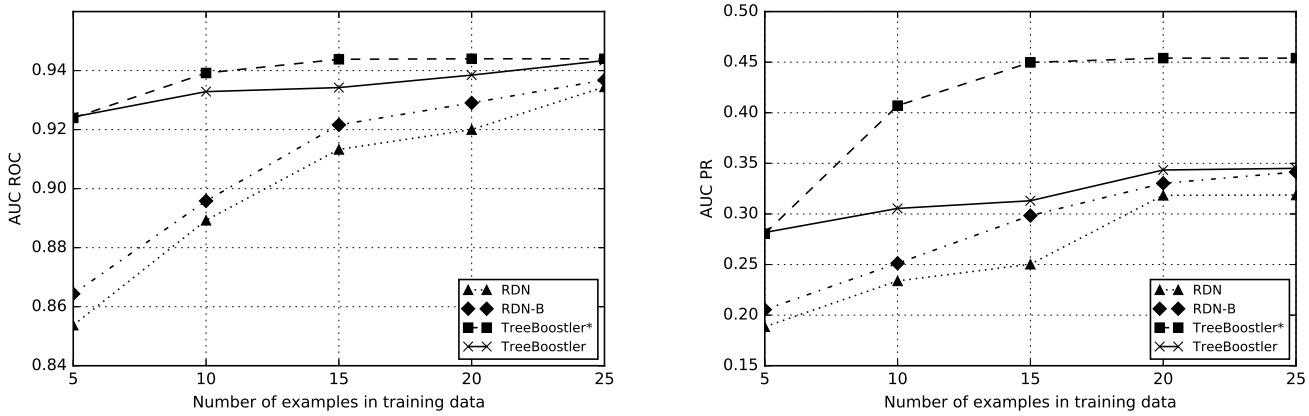

Fig. 7. Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from IMDB → UW-CSE.

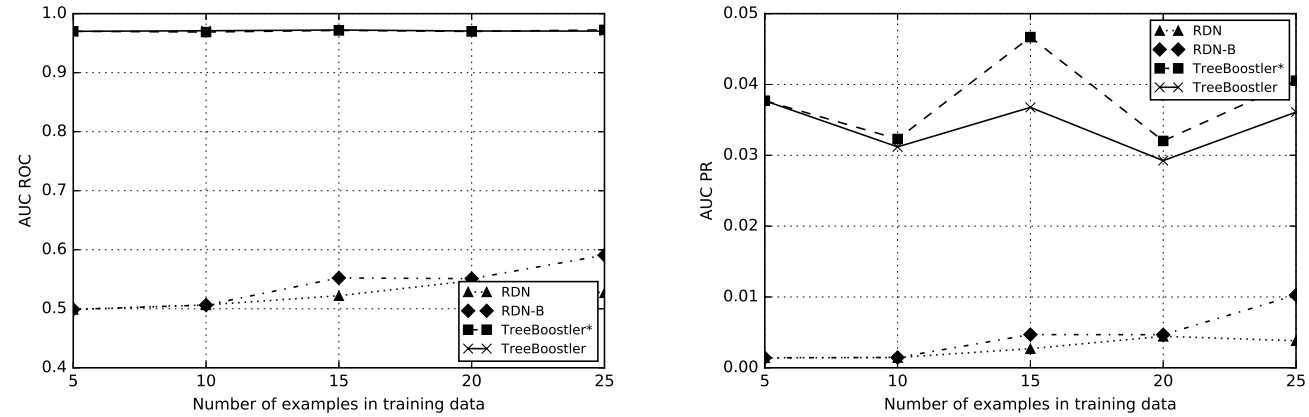

Fig. 8. Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from NELL Sports → NELL Finances.

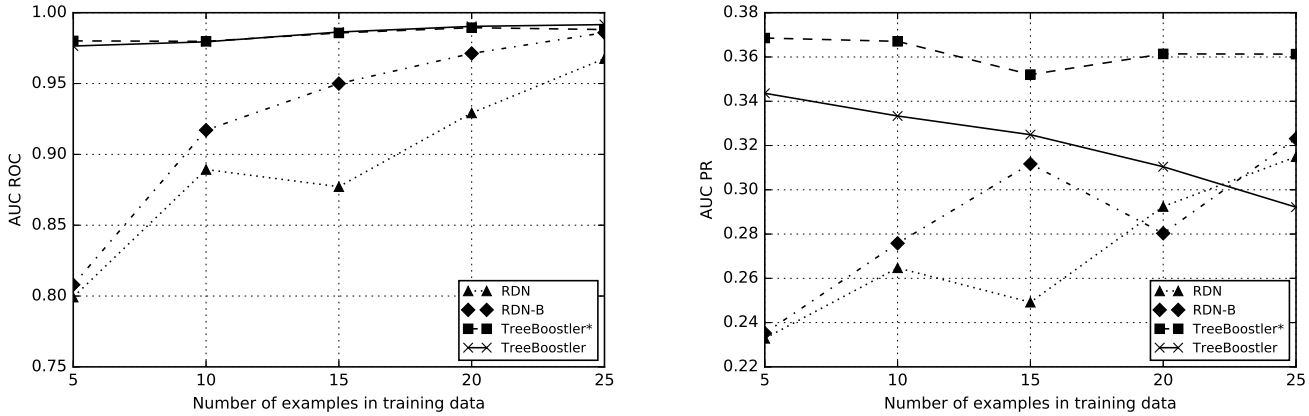

Fig. 9. Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Yeast → Twitter.

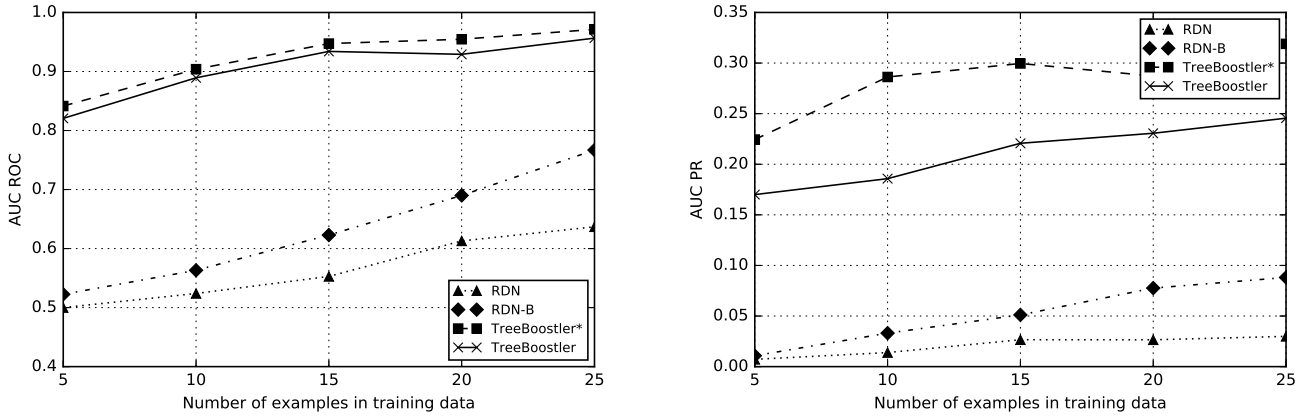

Fig. 10. Learning curves from minimal target data for AUC ROC (left) and AUC PR (right) obtained from Twitter → Yeast.

the experiments by simulating a transfer learning scenario where only a few data are available. We showed from experiments that transfer learning may result in much more accurate models compared to learning from scratch methods, however, it may also hurt the learning performance and result in less accurate models. Moreover, the theory revision process improved the performance of the transferred models showing the effectiveness of proposing modifications to fit the model to the target data. However, the experiments also showed that theory revision is more time consuming since it is another search approach. According to the experiments, our algorithm demonstrated to be as much efficient as learning from scratch methods.

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.

REFERENCES

[1] K. Lee, J. Caverlee, and S. Webb, "Uncovering social spammers: social honeypots+ machine learning," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 435–442.

[2] J. Sinapov and A. Stoytchev, "Object category recognition by a humanoid robot using behavior-grounded relational learning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 184–190.

[3] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," *CoRR*, vol. abs/1610.04286, 2016.

[4] J. Pan, W. Zheng, Q. Yang, and H. Hu, "Transfer learning for wifi-based indoor localization," in *AAAI 2008*, 2008.

[5] J. Blitzer, M. Dredze, and F. Pereira, "Biographies, bollywood, boomboxes and blenders: Domain adaptation for sentiment classification," in *In ACL*, 2007, pp. 187–205.

[6] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, no. 10, pp. 1345–1359, 2010.

[7] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 193–200.

[8] J. w. Lee and C. Giraud-Carrier, "Transfer learning in decision trees," in *2007 International Joint Conference on Neural Networks*, Aug 2007, pp. 726–731.

[9] H. W. Mewes, K. Heumann, A. Kaps, K. Mayer, F. Pfeiffer, S. Stocker, and D. Frishman, "MIPS: a database for genomes and protein sequences," *Nucleic Acids Research*, vol. 27, no. 1, pp. 44–48, 1999.

[10] J. Van Haaren, A. Kolobov, and J. Davis, "TODTLER: Two-order-deep transfer learning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI Conference on Artificial Intelligence*, 2015.

[11] L. Getoor and B. Taskar, *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[12] J. Davis and P. Domingos, "Deep transfer via second-order markov logic," in *Proceedings of the 26th International Conference on Machine Learning (ICML-09)*, 2009.

[13] M. Richardson and P. Domingos, "Markov logic networks," *Mach. Learn.*, vol. 62, no. 1-2, pp. 107–136, Feb. 2006.

[14] L. Mihalkova, T. Huynh, and R. J. Mooney, "Mapping and revising markov logic networks for transfer learning," in *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1*, ser. AAAI'07. AAAI Press, 2007, pp. 608–614.

[15] L. Mihalkova and R. Mooney, "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, July 2009, pp. 1163–1168.

[16] R. Kumaraswamy, P. Odom, K. Kersting, D. Leake, and S. Natarajan, "Transfer learning via relational type matching," in *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ser. ICDM '15. IEEE Computer Society, 2015, pp. 811–816.

[17] S. Natarajan, T. Khot, K. Kersting, B. Gutmann, and J. Shavlik, "Gradient-based boosting for statistical relational learning: The relational dependency network case," *Mach. Learn.*, vol. 86, no. 1, pp. 25–56, Jan. 2012.

[18] S. Wrobel, "First order theory refinement," in *Advances in inductive logic programming*. IOS Press, 1996.

[19] A. Paes, G. Zaverucha, and V. S. Costa, "On the use of stochastic local search techniques to revise first-order logic theories from examples," *Machine Learning*, vol. 106, no. 2, pp. 197–241, 2017.

[20] A. L. Duboc, A. Paes, and G. Zaverucha, "Using the bottom clause and mode declarations in FOL theory revision from examples," *Machine Learning*, vol. 76, no. 1, pp. 73–107, 2009.

[21] H. Khosravi, O. Schulte, J. Hu, and T. Gao, "Learning compact Markov logic networks with decision trees," *Machine Learning*, vol. 89, no. 3, pp. 257–277, 2012.

[22] L. Mihalkova and R. J. Mooney, "Bottom-up learning of markov logic network structure," in *Proceedings of 24th International Conference on Machine Learning (ICML-2007)*, 2007.

[23] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, ser. AAAI'10. AAAI Press, 2010, pp. 1306–1313.